An Efficient Framework for the Manipulation of OLAP Hierarchies

Ahmad Taleb

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March, 2007

Your file Votre référence
ISBN: 978-0-494-28956-3
Our file Notre référence
ISBN: 978-0-494-28956-3

ABSTRACT

An Efficient Framework for the Manipulation of OLAP Hierarchies

Ahmad Taleb

Online Analytical Processing (OLAP) is a database paradigm that supports the rich analysis of multi-dimensional data. OLAP is often supported by a logical structure known as the Cube, a data model that provides an intuitive array-based perspective of the underlying data. However, supporting efficient OLAP query resolution in enterprise scale environments is an issue of considerable complexity. In practice, the difficulty of the problem is exacerbated by the existence of dimension hierarchies that sub-divide core dimensions into aggregation layers of varying granularity. Common hierarchy-sensitive query operations such as Rollup and Drilldown can be very costly. Moreover, facilities for the representation of more complex hierarchical relationships are not well supported by conventional techniques.

This thesis presents a robust hierarchy and caching framework that supports the efficient and transparent manipulation of attribute hierarchies within relational environments. Experimental results show that compared to the current methods, very little additional overhead is introduced by the proposed framework, even when advanced functionality is exploited.

ACKNOWLEDGEMENTS

It is really an honour and a privilege to express my gratitude and indebtedness to my supervisor, Todd Eavis for his priceless support, encouragement and inspiration. His rich experience, wealth of knowledge, and critical and creative thinking has given me direction and insight in pursuing this research.

Special thanks are extended to my uncle Dr. Nasser Taleb for his kindness, support and encouragement. I am really glad that I have an uncle like him in my life.

Also, I would like to take this opportunity to personally thank all of my colleagues and friends who have been so supportive and giving of their time especially Ali Kiani and Tanbir Ahmed.

Finally, I treasure the invaluable support and encouragement of my dear mother. I also owe my loving thanks to my wife Bouchar Taleb and my son Bahaa. They have endured a lot from my research and traveling. If not for their support, understanding and encouragement it would not be possible to complete this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Hierarchical OLAP Processing

Online Analytical Processing (OLAP) has become an important component of contemporary Decision Support Systems (DSS). Central to OLAP is the data cube, a multidimensional data model that presents an intuitive cube-like interface to both end users and DSS developers. In recent years, the academic community has become increasingly interested in the cube model and a number of efficient cube generation algorithms have been presented in the literature[1, 28, 35].

The focus of these algorithms has been on generation of the cube data structure. Methods or techniques for efficient accessing/querying have received relatively little attention. When such methods have been presented, they typically assume the existence of non-hierarchical attributes or dimensions. In practice this is rarely the case. Figure 1.1 provides a simple example from the automotive industry. Here, we have three "feature" attributes — Product, Location, and Time — that can be viewed in terms of one or more "measure" attributes. Specifically, the measure attribute represents a key organizational metric that is associated with a unique combination of feature values. In this case, each cell in the cube might provide an aggregated total

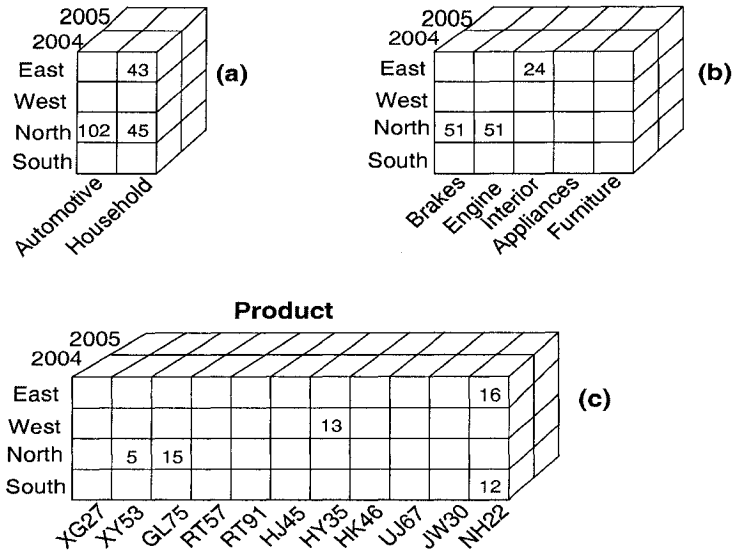


Figure 1.1: A Hierarchal Product attribute broken down from (a) category, to (b) type, to(c) product number.

for the measure attribute "Total Sales."

Given a product, a region, and a year, we have at most one Total Sales value. In figure 1.1(a), the total sales for the Automotive products in the North region in 2004 is 102. Note, however, how the hierarchical Product dimension on the x-axis is broken down into increasingly finer levels of aggregation. For example, the Automotive category is broken down into the Brakes division and the Engine division which, in turn, are broken down into the Product numbers XG27, XY53, GL75, and RT57. In practice, this type of attribute specialization is typical of most corporate environments.

While it is possible to represent each of these hierarchical levels as a distinct feature attribute, doing so dramatically increases the complexity of the underlying problem. Specifically, the data cube represents the views formed from all combinations of relevant feature attributes (i.e., the primary entities like Customer and Product).

Unfortunately, the number of possible views (or group-bys) in a d-dimensional data cube is exponential on the number of dimensions. For example, a 10-dimensional cube would generate $2^{10} = 1024$ aggregated group-bys.

So our basic Product-Location-Time data cube in Figure 1.1 would define 8 unique group-bys (that is, all the non-empty subset of {Product, Location, Time}). By contrast, the total number of group-bys in the presence of hierarchies is given as $\prod_{i=1}^{d}(h_i + 1)$ when constructed from a data cube with d attributes, where dimension i has a hierarchy of size h [32]. In our example, the total number of group-bys for the three-dimensional data cube Product-Location-Time with a hierarchy of size 3 on the Product dimension would produce 32 views (e.g., [Category-Location-Time], [Type-Location-Time], [Product-Location-Time], [Category-Location], [Type-Time], etc.). If this does not appear to be an obvious problem, consider that a 10-dimensional data cube with three-level hierarchies on each dimension would produce over one million group-bys!

Clearly this is infeasible when the original input set contains terabytes of data. An alternative approach to the generation and storage of fully materialized hierarchical cubes is to produce data cubes containing hierarchies represented only at the finest level of granularity. Manipulating hierarchies at non-materialized granularities is then done in real time during query processing. In this case, we embed the hierarchical attributes within conventional relational tables and subsequently employ standard SQL queries to process the appropriate grouping or aggregation results. However, while providing a clean interface for simple hierarchies, the SQL-based approach is not easily extendible because it possesses no native concept of dimensions or hierarchies, and treats the cube as just one or more standard tables. Moreover, extensive join

operations are typically required in order to fully resolve such hierarchical operations.

Given the limitations of the previous methods, another possibility is to extend the relational model to include hierarchy sensitive data structures. Without the restrictions imposed by a computational framework that was not explicitly designed to support complex hierarchies, the extended model offers the potential to support more powerful hierarchical logic. In order for this approach to be feasible, of course, the associated overhead should be largely transparent to the end user. To our knowledge, no such mechanism has been presented in the literature.

1.2 Transparent Mapping Methods

As noted above, current OLAP tools are quite limited in their ability to model attribute hierarchies. Specifically, as will be fully discussed in Chapter 2, real world hierarchies come in many forms, some of which can be quite difficult to support easily or efficiently. In this thesis, we present a number of algorithms and data structures for the efficient manipulation of realistic attribute hierarchies in "real time." We begin with a structure called the hMap that has been designed for the processing of what is known as *strict symmetric* hierarchies[9]. These are the simplest hierarchy form and are supported by some commercial tools. Nevertheless, the hMap provides a clean and efficient processing model that highlights the main features of our general approach.

We then present an extended structure known as the xMap that can be used for the kinds of practical, real world hierarchies that are not easily or efficiently handled by existing techniques. We use xMap to deal with both symmetric hierarchies and what are known in the field as *ragged* hierarchies. Here, hierarchy paths may have

alternate nodes. Using the conceptual approach of the hMap, we construct a more flexible tree-based data structure that is far more efficient than alternate techniques.

Finally, we extend the core model with a third structure known as the nMap that can be used to handle *non-strict* hierarchies, or those having many-to-many relationships between levels. These are by far the most difficult hierarchies to model efficiently and, in fact, there are few if any tools available to conveniently model this form of hierarchy in practice. Again, our nMap structure is both computationally efficient and conceptually straightforward.

Though we describe three separate data structures, it is important to note that our hierarchical models represent a single, coordinated framework. Based upon the specific requirements of the underlying data model, the associated query engine can choose the most appropriate solution. In fact, as we will demonstrate, the more sophisticated xMap and nMap can actually be integrated into a common mapping graph.

1.3 Experimental Results

Because it is important to ground performance sensitive query frameworks within workable, fully functionally database prototypes, we have integrated the new structures into the Sidera ROLAP Server. Sidera is the server component of the larger cgmCube Project [4, 7], a research platform designed to support terabyte scale data cubes. Our experimental results demonstrate that not only are the storage requirements quite modest but that real time processing overhead is likely to be imperceptible to the end user. We have extensively evaluated the overhead associated with our data structures for hierarchical attribute transformation. The average total overhead is less

than 12% for the hMap, 17% for the xMap, and 39% for the nMap. We have also compared our work with both production systems and alternative techniques from the research literature. Test results clearly indicate a significant performance advantage of our approach. Finally, we provide experimental support for the hierarchy-aware caching framework that has been integrated into the core Sidera query engine.

1.4 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 provides an overview of Online Analytical Processing, including concepts of fundamental OLAP operations and server architectures. The chapter also presents a classification of hierarchies in the real-world and a discussion of previous work in the area.

The succeeding chapters present the core contributions of the thesis, including the proposed algorithms, the implementation issues, and the experimental results. Chapter 3 describes our new data structures and algorithms (hMap, xMap, and nMap). It also discusses the algorithms used by the Sidera parallel query engine when resolving hierarchical queries, as well as the integration of our new methods into this environment. In chapter 4, our experimental results are presented. Finally, in chapter 5, we offer conclusions and briefly describe possible future work.

Chapter 2

Background Material

2.1 Introduction

Enterprise systems are becoming increasingly more complex. Organizations today have a mixture of older, centralized systems and newer, distributed systems; a wide variety of technologies is provided by an even larger number of vendors. Faced with this environment, IT departments have started to develop new concepts and tools for managing information technologies, as well as processing the wealth of data and information generated by them.

In this chapter, we examine the current trends, technologies, and terminologies related to an understanding of Online Analytical Processing or OLAP[3, 5, 8].

In section 2.2, we provide an introduction to decision support systems and their primary components: information processing, OLAP, and data mining. Section 2.3 defines On-Line Analytical Processing. In section 2.4, we discuss the data warehouse and the data cube. Section 2.5 presents a classification of hierarchies in the real world, and section 2.6 discusses previous work. Section 2.7 concludes the chapter with a brief summary.

2.2 Decision Support Systems

Decision Support Systems (DSS) are a specific class of computerized information system that supports business and organizational decision-making activities. A properly designed DSS is an interactive software-based system intended to help decision makers compile useful information from raw data, documents, personal knowledge, and/or business models to identify and solve problems and make decisions. Below we explain the main DSS models, including OLAP which is the focus of our research.

- **Information Processing.** Here we focus on the fundamental querying and reporting functions. Information processing systems accepts queries —whether ad-hoc or pre-defined— and processes data to provide information to decision makers. At this point, we only need a simple analysis, consisting of extracting, projecting, sorting, and basic aggregation.

- **OLAP.** Online Transaction Processing extends the basic capabilities of the Information Processing systems by allowing us to answer analytical queries that are multi-dimensional in nature. OLAP is useful to work with the data warehouse (to be discussed in section 2.3) and with a set of OLAP tools. OLAP tools enable users to analyze different dimensions of multidimensional data from a variety of perspectives and hierarchies. OLAP functionality may include operations for calculations and modeling applied across dimensions, through hierarchies and/or across members, trend analysis over sequential time periods, and analysis of historical and projected data in various "what-if" data model scenarios.

- **Data Mining.** This is a term used to describe knowledge discovery in databases. It includes tasks such as knowledge extraction, data archaeology, data exploration, data pattern processing, and information harvesting. Data mining tends to be a data driven process while OLAP is driven by the user or the user's intention to verify his or her queries. Typical data mining operations include classification (infers the defining characteristics of a certain group), association (identifying relationships between events), and clustering (identifying groups of items that share a particular characteristic).

2.3 Defining OLAP

The term OLAP was coined in 1992, when E. F. Codd who first produced the relational data model in 1970, published a report entitled "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate"[5]. In this paper Codd indicated twelve points which make up any OLAP application. The following five points are the most important ones taken from his report:

1. Multidimensional conceptual view. Focus is the relationship between dimensions.

2. Transparency. The end user should not have to be concerned about details of data access or conversions.

3. Accessibility. OLAP should present the user with a single logical schema of the data.

4. Flexible reporting. Reporting must be capable of presenting data to be synthesized, or information resulting from animation of the data model according to

any possible orientation.

5. Unlimited dimensional and aggregation levels. A serious tool should support a big number of dimensions.

2.3.1 OLAP: A Functional Definition

While commercial OLAP systems may provide many functions, there is a minimal set that can and should be defined by any OLAP application. These functions are listed below, while graphical models are shown in figures 2.1, ??, and 2.2.

- **Pivot.** This OLAP operation allows users to re-organize the axes of the cube. Pivot deals with presentation. Figure 2.1 provides a simple example on how the pivot operation works in practice.

- **Slice.** This is an operation whereby we select a subset of a multi-dimensional array (or cube) corresponding to a single value for one or more members of the dimensions. This operation allows the user to focus in on values of interest. Figure 2.2 shows the process for a single value of the "color" dimension.

- **Dice.** The dice operation is a slice on more than two dimensions of a data cube (or more than two consecutive slices). The user can draw attention to desired blocks of aggregated data. In figure 2.2, we show a multi-dimensional subcube of a larger cube space.

- **Roll-up.** This is a specific analytical technique whereby the user navigates among levels of data ranging from the most detailed (down) to the most summarized (up) along a concept hierarchy. Figure 2.2 illustrates how the "color"

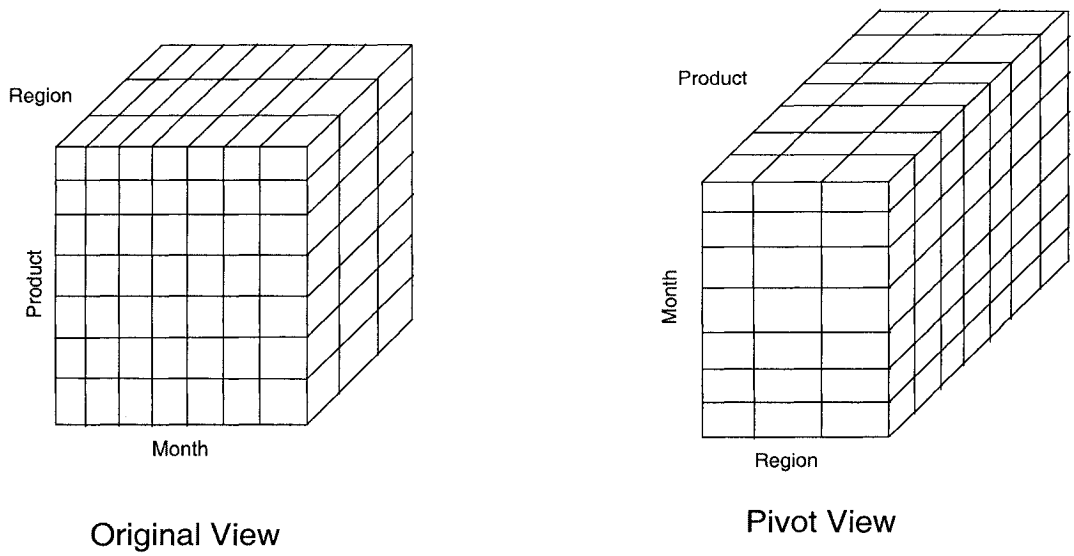


Figure 2.1: Pivot Operation

dimension, originally listed at a detailed level color, is aggregated in order to provide a break down by summer and winter colors.

- **Drill down.** This is a specific analytical technique whereby the user navigates among levels of data ranging from the most summarized (up) to the most detailed (down) along a concept hierarchy. Figure 2.2 shows how the "item" dimension is broken down to its item numbers.

Our objective in this thesis is to develop an efficient mechanism for the manipulation of these operartions within complex OLAP environements.

2.4 The Data Warehouse and Data Cube

The concept of the data warehouse begins with the physical separation of a company's operational and decision support environments. In other words, a data warehouse is a distinct corporate database management system (DBMS) that is designed to facilitate

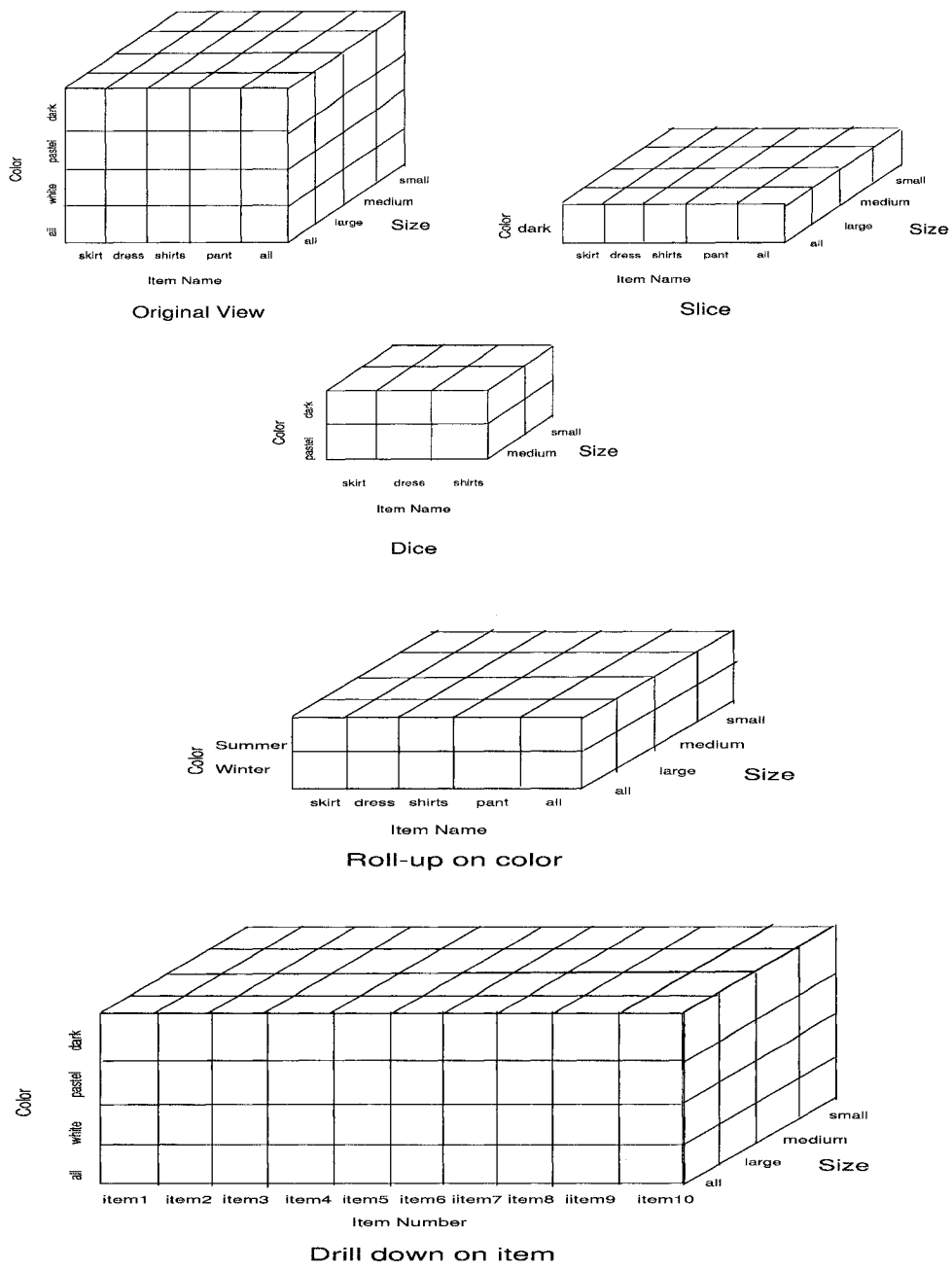


Figure 2.2: Slicing, Dicing, Roll-up and Drill-down on a simple three dimensional cube.

super fast queries times, as well as the analysis of multidimensional data. The data warehouse is the central data repository for virtually all OLAP systems.

So why do we need a separate database? Virtually all contemporary companies possess a vast store of operational data, usually derived from critical mainframe-based online transaction processing (OLTP) systems. Unfortunately, these systems are ill suited for decision making. Specifically, OLTP systems are designed to meet the day-to-day operational needs of the business, and database performance is tuned for those operational needs.

Given this brief description of the OLTP environment, we can say that the purpose of the data warehouse is to establish a data repository that makes operational data accessible in a form that is readily useable for decision support applications.

More formally, we borrow the definition in[16] of Bill Inmon who described the data warehouse as follows:

- **Subject oriented.** Data is organized so that all the data elements relating to the same real-world event or object are linked together.

- **Time variant.** Changes in the data are tracked and recorded so that reports can be produced showing changes over time.

- **Non volatile.** Data is never over-written or deleted, but retained for future reporting.

- **Integrated.** The data warehouse contains data from most or all operational applications of an organization, and this data is made consistent.

2.4.1 Data Warehouse Architecture

Data warehouses can be seen as a three-tier architecture [3, 13]. The canonical data warehouse architecture is shown in figure 2.3. The possible data sources are shown on the left. Information is extracted from various legacy systems and operational sources, consolidated, summarized, and loaded into the data warehouse. Strictly speaking, this first step is outside the scope of the warehouse proper. Several data marts are shown in the second stage, each of which is a small warehouse designed for a specific department. At this stage, we have the actual data warehouse, which contains the "decision support" data and associated software. We can refer to this component as the first tier. The second tier contains the OLAP server/engine that allows the users to access and analyze data in the warehouse. In practice, there are many forms of OLAP servers; they are used for the same aims but differ in their internal data representations. Finally, the third tier includes the front end tools that provide a graphical interface for the top managers and decision makers.

2.4.2 The Data Cube

The data cube is a multidimensional model that supports OLAP processing. It can be described as a data abstraction that allows one to view aggregated data from a number of perspectives. As mentioned in the introduction, a data cube consists of dimensions and measures. Dimensions are also known as attributes. Attributes can be of two types. Feature attributes represent entities, such as employee and product. Measure attributes refer to the items of interest. The measure attributes are aggregated according to the feature attributes. Examples will be discussed later in this section.

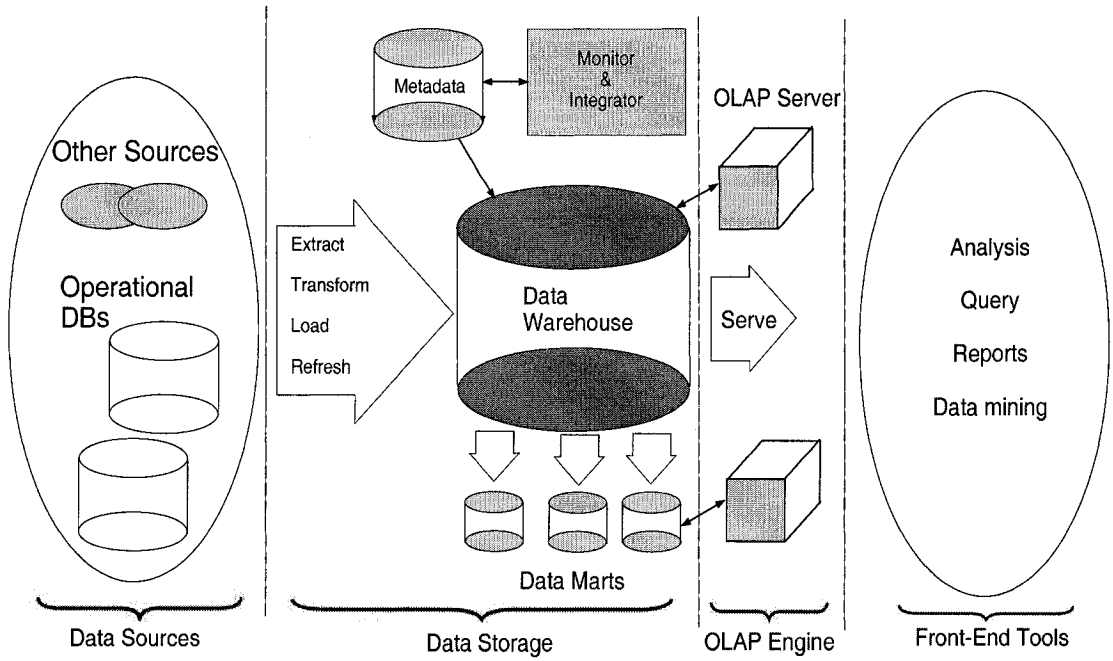
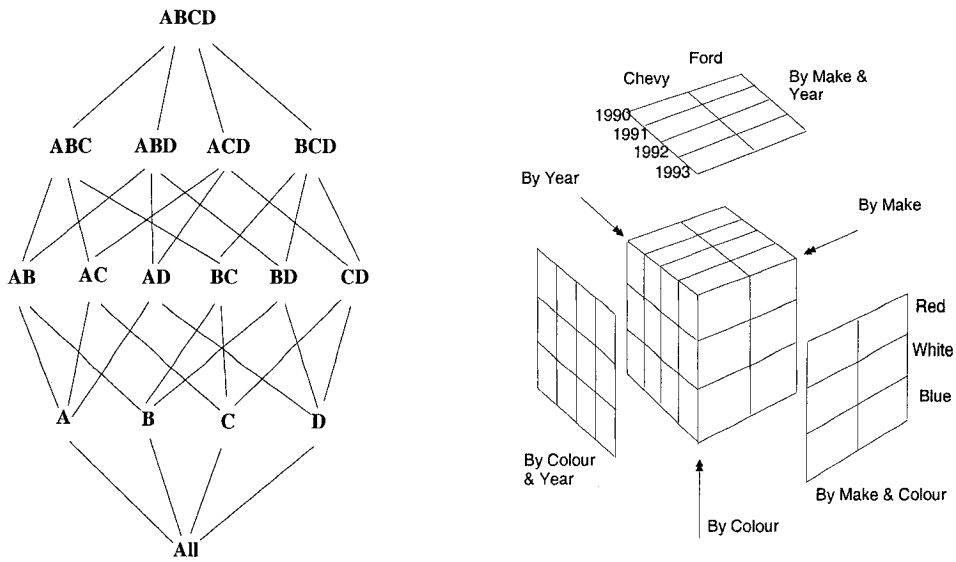


Figure 2.3: Data Warehouse Architecture

For a d-dimensional space $\{A_1, A_2, \ldots, A_d\}$, we have $O(2^d)$ attribute combinations. We often refer to this collection of views as the *power set*. We refer to the number of unique values in each of these d dimensions as the attribute cardinality $C_i, 1 \leq i \leq d$. The complete cube space is equivalent to the *cardinality product* $C_x = \prod_{i=1}^{d} C_i$. Large cardinality products are associated with sparse cube spaces. By sparse, we mean that the ratio N/C_x of actual records stored N to the cardinality product is relatively small (there is no agreed upon ratio to separate between sparse and dense spaces). In OLAP, views are also known as cuboids or group-bys. Each view represents a distinct combination of feature attributes, and can be seen as presenting an aggregation of the measure attribute.

Logically, the cube can be represented in a number of ways. To identify the relationships between the views in the cube Power Set, we typically use a *lattice*



(a) (b)

Figure 2.4: Two views of the set of data cube group-bys. (a) A four dimensional lattice (with single letters substituted for dimension names) and (b) A geometric depiction showing several perspectives for an automotive example.

representation. Figure 2.4(a) shows the lattice in a four dimensional cube space. Alternatively, we may view the cube in terms of a more intuitive geometric representation. In other words, the cube can be drawn as a hypercube, with the axes of the cube identifying the feature attributes, while the cell values identify the aggregated measure for a given combination of feature attributes. Figure 2.4(b) presents a simple example from the automotive industry.

The data cube consists of the base cuboid plus (2^d)-1 cuboids. Since the base cuboid contains all the feature attributes, it can be used to compute all the coarser cuboids by aggregating across one or more of its dimensions. The data cube can be described as "full" if it contains all the (2^d) possible views, or "partial" if only a

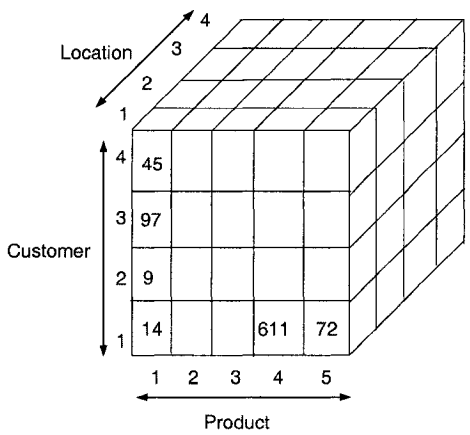
subset of views has been constructed.

While the cube can be defined as a logical data model, it often forms the basis of a physical model as well. Specifically the group-bys can be pre-computed and stored to disk to improve real time query performance. If the data is physically stored as a multi-dimensional array (as in figure 2.4(b)), we have what is called a MOLAP design. MOLAP provides implicit indexing along the axes of the multi-dimensional array but performance sometimes deteriorates as the space - and the associated cube array - becomes more sparse (high dimensionality/high cardinality). Relational Olap, or ROLAP, stores group-bys (view/cuboids) as distinct tables and tends to scale well since only those records that actually exist are materialized and stored. However, it requires explicit multidimensional indexing in order to be used effectively. Hybrid OLAP, or HOLAP, is an approach that tries to combine the best of both models by using MOLAP storage for dense regions of the lattice space and ROLAP storage for those views that have a greater degree of sparsity.

Figure 2.5 shows the construction of the base cuboid of a three dimensional cube (Product, Customer, Location). Figure 2.5(a) presents the MOLAP model, while figure 2.5(b) depicts the ROLAP case.

2.4.3 The Star Schema

The *star schema* is perhaps the simplest data warehouse design. It is called a star schema because the entity-relationship diagram of this schema resembles a star, with points radiating from a central table. The center of the star consists of a large *fact table* and the points of the star are the dimension tables. Here the heavily normalized schema of the OLTP environment is converted into a multi-dimensional model. A star schema is characterized by one or more very large fact tables that contain the



(a)

Product Key	Customer Key	Loaction Key	Measuer Attribute(s)
1	1	1	14
1	2	1	9
1	3	1	97
1	4	1	45
3	1	1	611
4	1	1	72

Product-Customer-Location Base cuboids

(b)

Figure 2.5: Two views of the base cuboids of a three dimensional cube. (a) MOLAP model and (b) ROLAP model.

primary information in the data warehouse, and a set of one or more de-normalized dimension tables, each of which contains information about the entries for a particular attribute in the fact table. By de-normalized, we mean that tables maintain some of the redundancy that good OLTP systems typically eliminate.

A star query is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The cost-based optimizer recognizes star queries and generates efficient execution plans for them. De-normalizing the dimension tables significantly reduces the number of expensive joins that would otherwise be required with a normalized schema. The dimension tables are typically quite small compared to the massive fact tables. Therefore, the redundancy introduced by the de-normalization is of little concern in most OLAP contexts. In situations where dimensions like Product may be very large, *snowflake* schemas normalize dimensions

to eliminate redundancy. That is, the dimension data can be grouped into multiple tables, instead of one large table. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance.

Figure 2.6 illustrates a simple star schema, one that includes Customer, Location, and a hierarchy-based Product dimension. Note that the de-normalized Product dimension has been constructed from a multi-table Product design that would be common in an OLTP setting. Specifically, Product, Category and Brand are now housed in a single table, rather than a three table model expected in a typical normalized design. Note that the star schema maps directly to both the geometric cube and the lattice based cube (See Figure 2.4). Specifically, the fact table effectively represents the base cuboids of the lattice, as well as the central, fully materialized sub-cube in the geometric representation.

2.5 Hierarchies

We have generally spoken in term of a dimension as though it were a simple attribute concept. Many, if not all, common business and scientific dimensions actually have a hierarchical structure. In an informal way, everybody is familiar with some hierarchical dimensions. For example we often think of the common Time hierarchy in terms of: hours, days, weeks, months, quarters, and years. Product hierarchies are also very common. In OLAP environments, the traversal of such "aggregation hierarchies" is perhaps the most fundamental query format. In fact, in order to create a cube in most ROLAP or MOLAP systems, we typically need to specify complex dimension hierarchies in order to allow the user to see data at different levels of detail.

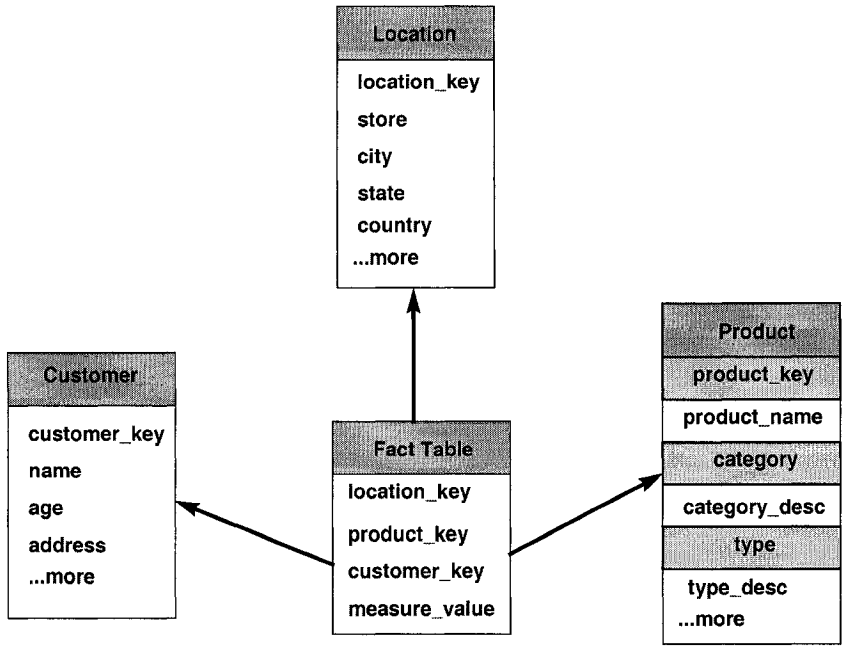


Figure 2.6: A simple Star shema showing a denormalized Product hierarchy.

2.5.1 Basic Terminology

We may describe a hierarchy as a set of binary relationships between the various levels of the dimension in the data cube. A *path* defines a unique traversal through a hierarchy from the *root* level to a *leaf* level. We consider the root to be the coarsest level of the hierarchy — often referred to as the "all" level — with the leaf identifying the finest level of aggregation detail. Within a given path, the nodes directly connected at two consecutive levels of the hierarchy are defined as the *parent* (above) and *child* (below). The values or instances at a given level of the hierarchy are known as members. The root is at level 0. In the following sections, we briefly classify the hierarchy forms commonly found in the real world; for the most part, this classification schema is modeled using or conforms to the framework defined by Malinowski et al. in [9, 10].

2.5.2 Simple Hierarchies

By simple, we mean any hierarchy that can be represented as a tree. Recall that a tree is a directed, acyclic graph. Simple hierarchies can be further divided into three basic categories:

1. Symmetric hierarchies.
2. Asymmetric hierarchies.
3. Generalized hierarchies.

2.5.2.1 Symmetric Hierarchies

Simple, symmetric hierarchies represent hierarchies with meaningful levels and branches that have a consistent depth. Symmetric hierarchies are also known as *homogeneous*, *balanced*, or *leveled-based* [9, 10]. Here, any path from the root to a leaf has exactly the same number of nodes. In other words, all nodes in the hierarchy tree are mandatory. Figure 2.7 shows a geographic hierarchy that has Country — Province — City — Store levels defined. The meaning and depth of each level is used consistently. They are consistent because each level represents the same type of information. For example at level 2 all values refer exclusively to cities. Also note that all paths from root to leaves in figure 2.7 are of length 4.

2.5.2.2 Asymmetric Hierarchies

A simple, asymmetric hierarchy is one in which lower levels of certain paths are not mandatory. Note that intermediate levels in the tree are mandatory. Several terms are used for these hierarchies: *heterogeneous*, *unbalanced*, and *non-onto* [9, 10]. Simple, asymmetric hierarchies are quite common in practice. Figure 2.8 shows a hierarchy

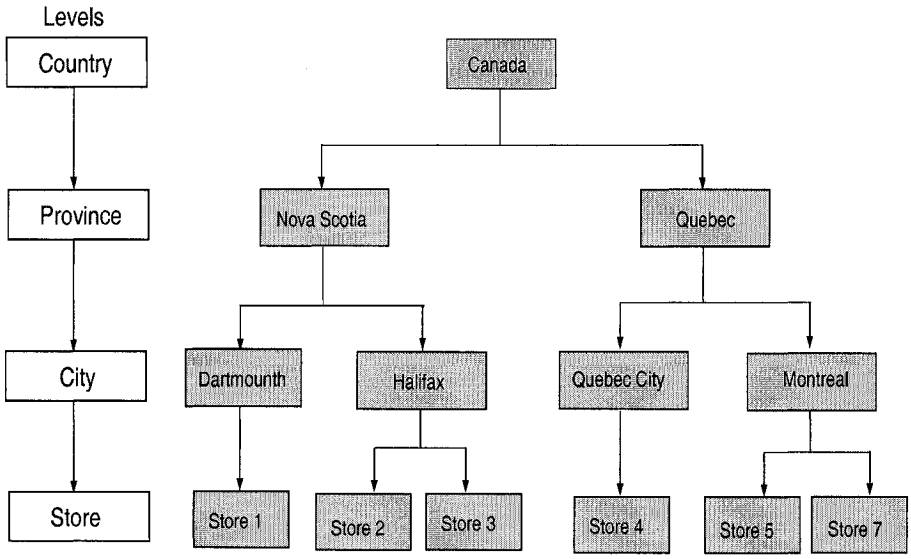


Figure 2.7: Example of a Simple, Symmetric Hierarchy

where a bank is composed of several branches: some of them have agencies with ATMs, some only agencies, and some branches do not have any agencies at all.

2.5.2.3 Generalized Hierarchies

The simple, generalized hierarchy is the most complex of the three simple forms. It can contain multiple exclusive paths that share some levels. In other words, different branches of the hierarchy tree appear to represent different types of things at the same level. The term *exclusive* implies that, given a leaf node, the path back to the root is uniquely defined. Figure 2.9 shows a generalized hierarchy tree that consists of two paths:

1. Area → Branch → Category → Profession → Customer

2. Area → Branch → Sector → Type → Customer

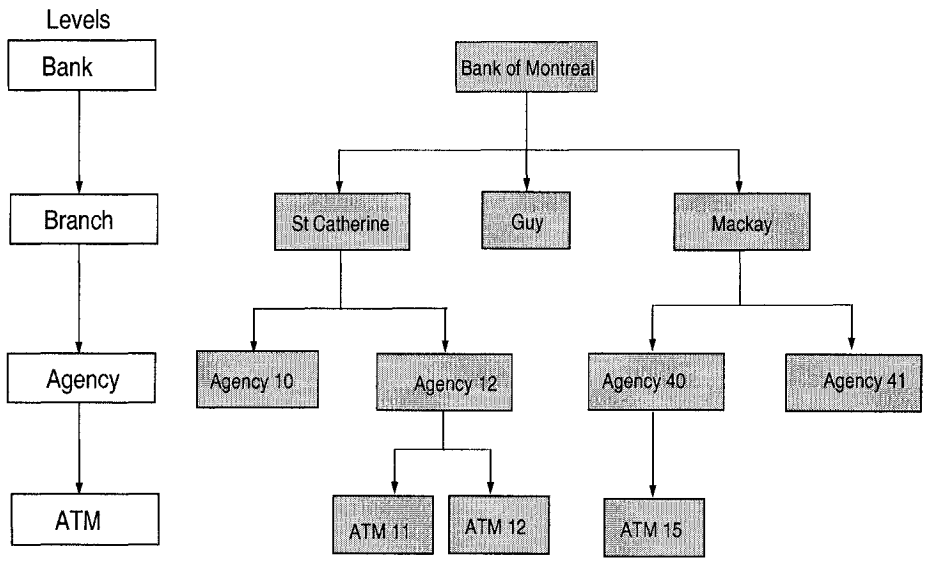


Figure 2.8: Example of a Simple, Asymmetric Hierarchy

In this case, the first path might refer to customers who are people, while the second path might refer to customers who are companies. Figure 2.9 illustrates that branches share the first, second, and fifth levels of the tree but the paths from the leaves are unique.

Of special significance is a particular form of the generalized hierarchy that can contain optional intermediate nodes. The branches have inconsistent depths because at least one intermediate member attribute in a branch level is unpopulated. It is known as the *simple ragged* or *simple non-covering hierarchy* [9, 10]. In some sense, the ragged hierarchy is like a cross between an unbalanced hierarchy and the regular generalized hierarchy. Figure 2.10 represents a sales company having stores in different countries. It shows that some provinces skip the county level. So we may have two valid paths like:

1. Country → Province → County → Store

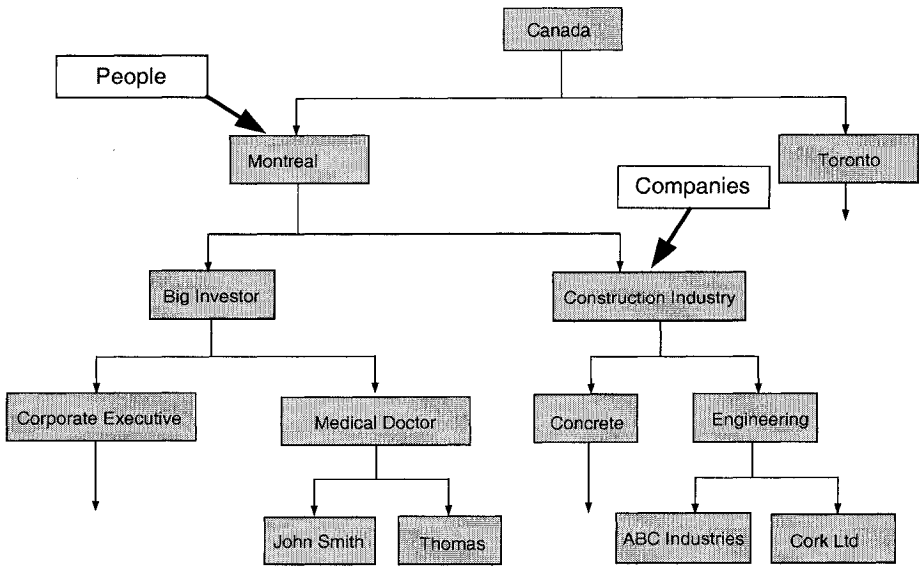


Figure 2.9: Example of a Simple, Generalized Hierarchy

2. Country → Country → Province → Store

We will use the terms ragged hierarchy and unbalanced hierarchy in the succeeding sections.

2.5.3 Strict versus Non-Strict

We call a hierarchy *strict* if one-to-many relationships exist between parent and child nodes. If many-to-many relationships exist between parents and children we call this type of hierarchy *non-strict*. Non-strict hierarchies are very common in real life applications, e.g. an employee could belong to more than one department. Note that it is possible for the simple hierarchies discussed so far to be either strict or non-strict. Figure 2.11 shows a non-strict hierarchy with 4 levels: Region → Division → Department → Employee. We have many-to-many relationships between the Department and Employee levels, but one-to-many relationships for others levels. Each

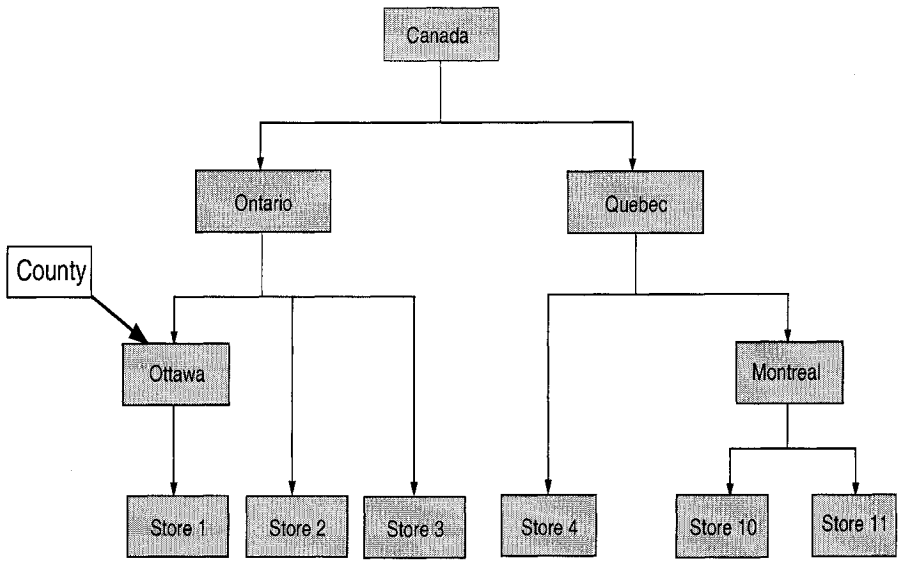


Figure 2.10: Example of a Ragged Hierarchy

department has many employees and each employee belongs to many departments, e.g. John Smith belongs to three different departments in figure 2.11.

2.5.4 Complex Hierarchies

In contrast to simple hierarchies, we have complex hierarchies. In effect, complex hierarchies represent combinations of simple hierarchies on a single dimension. We will look at two similar but distinct forms:

1. Multiple hierarchies.

2. Parallel hierarchies.

2.5.4.1 Multiple Hierarchies

In a multiple hierarchy, there are several non-exclusive simple hierarchies sharing some levels. Moreover, all such hierarchies share the root level. Multiple hierarchies

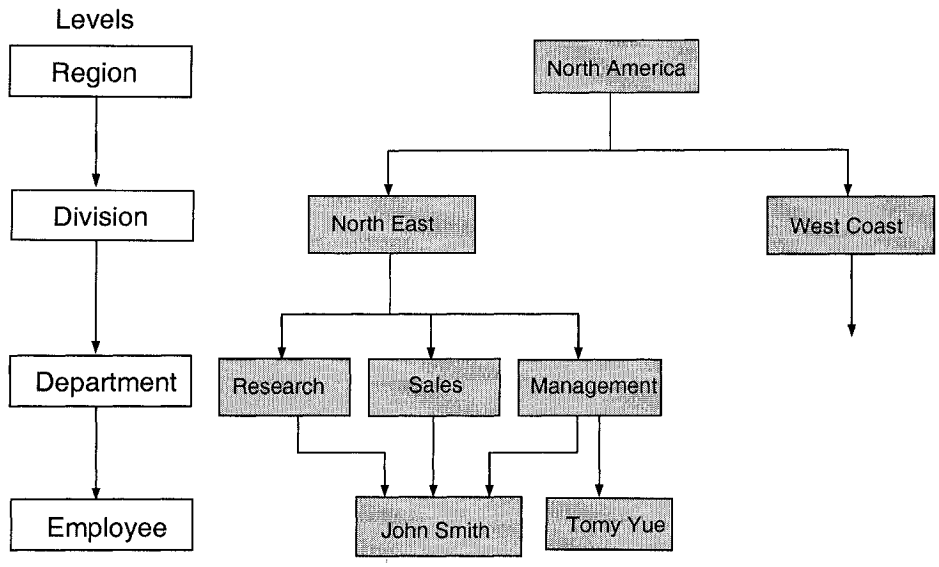


Figure 2.11: Example of a Non-strict Hierarchy

look similar to generalized hierarchies except that paths in multiple hierarchies are *non-exclusive*. By non-exclusive, we mean that it is possible for a leaf node to belong to more than one path. Figure 2.12 presents the Time dimension which includes two hierarchies corresponding to different calendar subdivisions: Year → Quarter → Month → Day and Year → Week → Day. The two hierarchies share the root level (year) and the leaf level (Day). A specific day, e.g. Day 1, can belong to two different paths.

2.5.4.2 Parallel Hierarchies

A parallel hierarchy is a collection of distinct simple hierarchies defined on the same dimension. As we have seen before, the simple hierarchies may be symmetric, asymmetric, etc. Unlike multiple hierarchies, parallel hierarchies do not share the root level. For this reason, parallel hierarchies are generally simpler than multiple hierarchies. Figure 2.13 shows an example of parallel hierarchies that share the leaf level

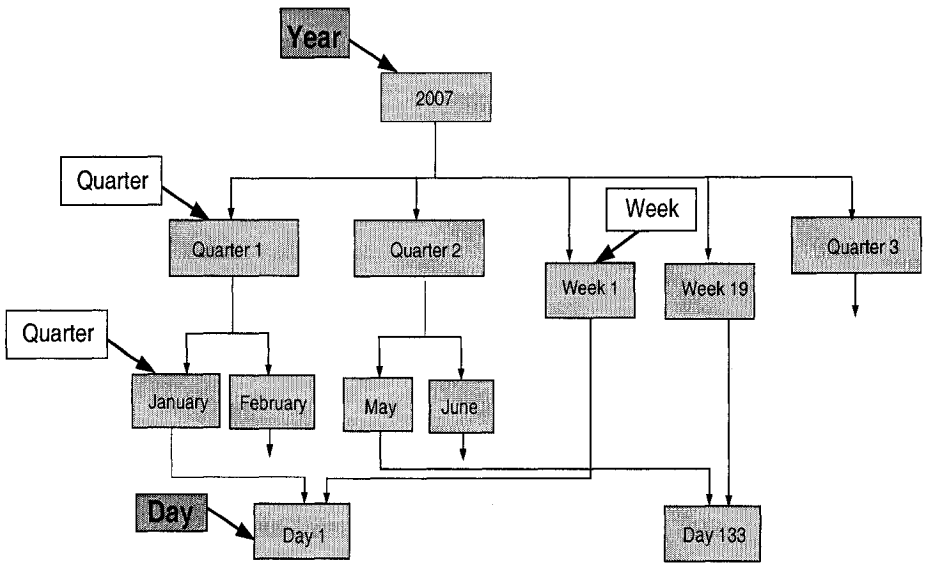


Figure 2.12: Example of a Multiple Hierarchy that shares the root level (Year) and the leaf level (Day)

(Store). The figure presents two valid parallel hierarchy paths:

1. Country → province → city → store

2. Sales Region → Sales District → store

2.6 Related Work

The data cube model was first introduced by Gray et al. [11]. In the succeeding years, a series of algorithms for the efficient computation of the data cube were presented. Most were based in some way upon the cube lattice presented by Harinarayan et al. [14] that identified the relationships between group-bys sharing common attributes. Academic research has generally favored the relational or ROLAP approach, in which group-bys are stored in conventional table format. Zhao et al.[35] have also

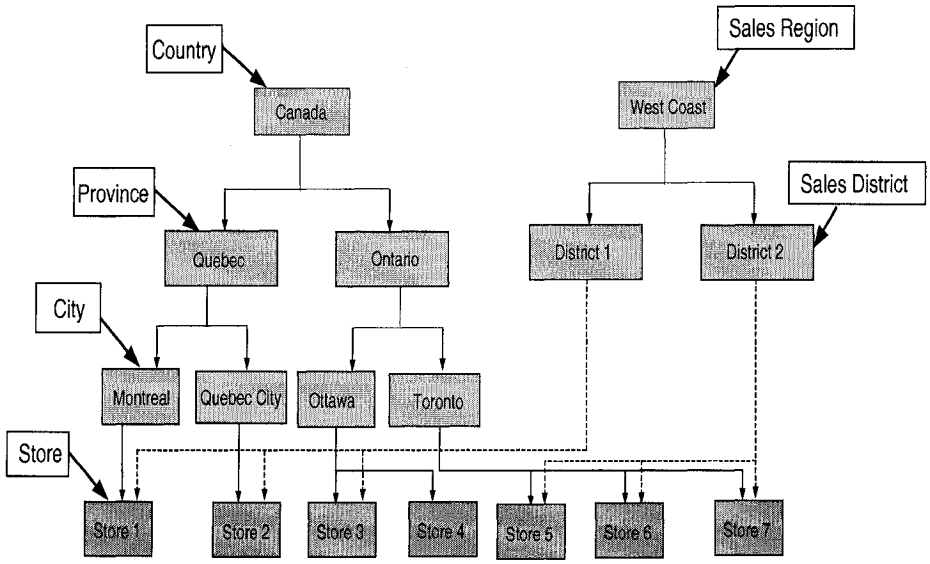


Figure 2.13: Example of a Parallel Hierarchy that shares the leaf level (Store)

presented an array-based algorithm. Though very efficient for dense, low dimensionality/cardinality data, this multi-dimensional or MOLAP model may not be scalable for large and sparse problem spaces.

Cabibbo and Torlone in [2] propose a design model for multidimensional databases. Hurtado et al. [15] build on that model. Their assumption of explicit availability of rollup functions between successive levels of dimension hierarchy suggests that query evaluation in their model is more likely to resemble that in the snowflake schema model.

Shukla et al. discuss the use of storage estimates for fully materialized hierarchies [32]. Sismanis et al. [33] propose a non-relational tree-based cube structure that eliminates prefix and suffix redundancies to create a cube data structure that is both compressed and searchable along attribute hierarchies. It is not clear, however, how amenable this structure is to complex range queries (as opposed to point

queries) or the parallelization and external memory requirements of enterprise-scale data warehouses.

Interestingly, a number of papers have discussed cube hierarchies in the context of conceptual modeling. Though the focus here is not upon the design or implementation of hierarchy sensitive data structures, a key benefit of this research has been a thorough treatment of hierarchy classification. Malinowski and Zimanyi provide a rigorous treatment of real world hierarchy requirements, and provide an ER-motivated notation for modeling each pattern [9, 10]. A more formal discussion of hierarchy dependency relationships is presented by Niemi et al. [26], with an emphasize on the underlying expressive power of the dependencies. The notion of summarizability in OLAP and statistical databases is discussed by Lenz and Shoshani [20], which focus on the elimination of inconsistencies in complex hierarchies that cannot be reliably aggregated.

A number of researchers have also discussed the impact of OLAP hierarchies on the design of efficient query languages. Jagadish et al. [17] examine extensions to SQL — SQL(H) — that would allow hierarchies to be manipulated at query time with a minimum of complexity (both in terms of query expression and execution cost). A hierarchy-aware algebra for multidimensional data is presented by Pedersen et al. [27] (along with a comparison of some of the early conceptual models). The authors also propose a mechanism for implementing non-strict hierarchies.

Commercially, various products offer their own proprietary hierarchy-aware languages. The most obvious of these is arguably Microsoft's MDX (Multidimensional Expressions), a component of its OLE DB specification [22]. In practical settings,

a number of common techniques have been used for storing and manipulating simple hierarchies in relational data warehousing environments. Kimball and Ross [19] discuss the use of de-normalized dimension tables in conjunction with the common Star Schema. For more complex hierarchies, Kimball and Caserta suggest the use of *embedded trees* and *bridging tables* [18].

2.7 Conclusions

The significance of sophisticated data analysis has grown enormously in recent years. Very often a data warehouse is used to support the process of data analysis, since it represents a large consistent repository for enterprise-wide corporate information. In turn, OLAP systems allow users to manipulate the data contained in the data warehouse. Within the OLAP model, dimension hierarchies are used to view and assess data at different levels of granularity.

In this chapter, we have examined the concept of Online Analytical Processing and the data cube. We began by reviewing the general area of decision support systems and its primary components, Information systems, OLAP, and data mining. OLAP operations were illustrated, along with explanations on how these operations are performed so as to provide meaningful measures of summarized multi-dimensional data. We have also examined the data warehouse architecture as a three tiered model and discussed the various kinds of OLAP servers: ROLAP, MOLAP, and HOLAP.

We then presented the notion of attribute hierarchies and defined different forms of hierarchies typically encountered in practical environments. This hierarchy classification forms the basis of our work with respect to that computational framework which will be further discussed in chapters 3 and 4.

Chapter 3

ROLAP Hierarchy Data Structures

3.1 Introduction

One of the more important research problems in the area of Decision Support Systems is the efficient manipulation of hierarchical dimensions stored in the data warehouse, thereby improving the efficiency of querying multi-dimensional data [32, 33, 9, 10, 26, 20, 17, 27, 22, 19, 18, 2, 15, 30]. As we have seen in section 2.6, most of the research that has been conducted in the area of OLAP hierarchies is focused on the theoretical treatment of hierarchy dependency relationships [9, 10, 26, 2, 15, 30, 32]. By contrast, little research effort has focused upon the manipulation of simple and complex hierarchies in Relational Online Analytical Processing (ROLAP) at query run-time. Existing methods typically exhibit poor query performance when dimensions are joined to a large fact table, are difficult to design, and/or require complex query logic to manipulate [17, 27, 19, 18]. Commercially, software vendors have developed products and services for improving the efficiency of at least a subset of hierarchical queries on data warehouse tables. Examples these products are Oracle 9i and Microsoft MDX (Multidimensional Expressions), a component of its OLE DB specification [22].

In this chapter, we describe in detail the design of three ROLAP hierarchy data structures (hMap, xMap, and nMap) that permit the efficient processing of symmetric strict hierarchies, ragged strict hierarchies, symmetric non-strict hierarchies, and ragged non-strict hierarchies. In general, our approach is to build efficient in-memory data structures that support specific hierarchy forms, and then to supply fast translation between arbitrary levels of the dimension hierarchy at run-time.

The chapter is organized as follows. In section 3.2, we present related techniques in the area of querying multidimensional data in the presence of hierarchies. Section 3.3 discusses the motivation of our own work. In section 3.4, we briefly describe preliminary material which includes: the basic parallel ROLAP architecture, the original query engine used to answer multidimensional queries, and some notation to support the presentation of our hierarchy data structures. A detailed description of the new hMap approach is provided in section 3.5. In section 3.6, we explain the xMap data structure in detail, while the nMap is presented in section 3.7. Section 3.8 explains how computed results on sub-attributes are cached so as to avoid re-computation whenever possible. We also describe the software model architecture. In addition, we illustrate in this section the query engine used in the Parallel ROLAP architecture to answer hierarchical queries. Section 3.9 is a review of the chapter objectives, with final conclusions provided in section 3.10.

3.2 Related Work

A significant amount of research has focused on foundation for the conceptual model for hierarchies in the data warehouse. Manipulation of different forms of hierarchies at run-time has received less attention. In this section we review the most significant

methods previously presented in the literature, and identify the benefits provided by these approaches, as well as their weaknesses. In addition, we identify the objectives of our research.

Some of the most significant work on manipulating hierarchies for efficient hierarchical query resolution has been reported by Kimball [19, 18]. Here, the author discusses a pair of approaches which model simple and complex hierarchies to support hierarchal queries in ROLAP systems. The first is known as the *recursive pointer* technique. This mechanism builds on how the nodes of a dimension hierarchy tree are related to each other. If we have a dimension hierarchy such as Customer, we clearly need enough information to show how these separate customers are related. A Parent Key field is added to the Customer dimension. The Parent Key field would be a recursive pointer that would contain the proper key value for the parent of any given customer. A special null value would be required for the topmost Customer in the enterprise. Although this simple recursive pointer method allows representing an arbitrary organizational tree structure of any depth [19, 18], the solution introduces a number of problems. The most important of which is that queries may exhibit poor performance when the dimensions are joined to a large fact table (i.e. the kind that Sidera is designed to support), and that one cannot use a recursive pointer technique in standard SQL to join the dimension table with the fact table.

Instead of using a recursive pointer, Kimball describes another technique known as *bridging tables* that can be used to solve this modeling problem by inserting a "helper" table between the dimension table and the fact table. In this case, we don't have to make any changes to either the dimension table or the fact table. However, this technique requires the design of additional tables that attempt to isolate the

hierarchy architecture from the remaining dimensional attributes. For the hierarchical customer dimension, for example, each record in the helper (bridge) table contains the following fields:

- Parent Customer Key
- Subsidiary Customer Key
- Depth From Parent
- Lowest Flag
- Topmost Flag

If descending through the tree from certain selected parents to various subsidiaries, we join the dimension table to the bridging table and the bridging table to the fact table. The Depth from Parent field counts how many levels the subsidiary is below the parent. The Lowest Flag field is true only if the subsidiary has no further nodes beneath it. The Topmost Flag field is True only if the parent has no further nodes above it.

The bridge table is used to implement non-strict hierarchies that have many-to-many relationships between two consecutives levels. Current OLAP tools do not manage this type of hierarchy. As such, it includes information about the measure distribution between levels. However, while the bridge table solves some of the problems introduced by the recursive technique, it still requires additional join operations, and is both difficult to design and necessitates additional complex query logic to manipulate. In large hierarchies, this helper table may become infeasible in that it may contain more records than there are nodes in the hierarchy tree. Nevertheless, we

emphasize that the most important feature of the bridging table is that it underlines the importance of cleanly separation of the hierarchical structure from the dimension table itself.

Another common implementation solution proposed by Pedersen [27] transforms non-strict hierarchies into independent dimensions. Here, the fact table contains more detailed data than that of the bridge table. Note, however, that with this method, the problem of double counting the same members may occur [27].

A number of other research papers [32, 30] treat the hierarchy levels as additional attributes, in which case the number of views to be materialized grows dramatically (refer to section 1.1). For example, the authors in [32, 30] deal with hierarchies in this manner suggesting to pre-compute hierarchical summaries during the data cube generation phase. However, this approach is extremely expensive in of time and space.

In [17], Jagadish et al. examine extensions to SQL – SQL(H) – that would allow hierarchies in relational data warehouses to be manipulated at query time with minimum complexity, particularly in terms of SQL programming effort. In order to do so, the authors decompose dimension hierarchies into arbitrary and possibly very large set tables for each level in the hierarchy(ies). But while the model of the associated extension produce a cleaner design, no experimental results to validate the approach were ever produced.

Commercially, Oracle has traditionally used something called OLAP DML (Data Manipulation Language) to improve aggregation performance in the data warehouse. It provides several extensions to the standard SQL GROUP BY clause to make query reporting faster and easier. The most important extension dealing with the hierarchal dimensions is the ROLLUP. ROLLUP calculates aggregate functions such as SUM,

COUNT, MAX, MIN, and AVG at increasing levels of aggregation, from the most detailed up to a simple total. This mechanism is very helpful for subtotaling along hierarchical dimensions such as time or geography in that it creates subtotals that roll up from the most detailed level to a grand total, as per a grouping list specified in the ROLL UP clause [34]. Techniques for more complex queries, such as those involving non-strict hierarchies are not provided.

Microsoft [22, 23] also offers its own proprietary hierarchy aware language, MDX which stands for Multidimensional Expressions. It supports the definition and manipulation of multidimensional objects and data. Though similar to SQL, MDX is not an extension of the SQL language. Specifically, MDX is intended to be a native OLAP query language. Just as the result of an SQL query is a table, the result of an MDX cube query is another cube. Hierarchies are directly supported in MDX by using the *dot notation.* For example, if we have a Time hierarchy with years, quarter, and month, we might use the notation [Time].[1998].[Q1] to specify the first quarter of 1998. MDX provides a number of functions for traversing hierarchy trees. Perhaps the most important function is *Children* which selects the set of all instances below the current instance. Other functions include Ancestor, Descendents, Siblings, and Cousin [22, 23].

MDX and Oracle's DML are the most famous OLAP query languages. However, as noted, they support only a subset of the hierarchies found in the real-world.

3.3 Motivation

Though the research efforts described in the previous section have attempted to model dimension hierarchies and to make extensions to the SQL query language to support

hierarchical ROLAP queries, they have been only a partial success. These efforts make it somewhat difficult to design dimension hierarchies, and require complex query logic to manipulate, or an extended SQL query language whose performance has not been verified. In fact, for all these techniques, it is likely that scalability would be quite poor in the case of large fact tables (i.e, the kind that Sidera is designed to support). For these reasons – complexity and performance – there is clearly an opportunity for further research with respect to querying and defining hierarchical ROLAP systems.

The following primary objectives are identified with respect to the new data structures that are required to model and query complex real world hierarchies:

1. Identify a simple design that is easy to implement and update.
2. Ensure minimal memory requirements.
3. Maintain the notion of *attribute linearity* (discussed in the succeeding sections).
4. Support efficient translation between arbitrary levels of the hierarchy at query run-time.
5. Support very fast queries for different kinds of hierarchical models.
6. Support the following four types of hierarchies: Symmetric Strict Hierarchies, Symmetric Non Strict Hierarchies, Ragged Strict Hierarchies, and Ragged Non Strict Hierarchies.
7. Provide a multidimensional hierarchical caching framework to support fundamental query types.

In the reminder of this chapter, we present details of our new approach that supports the efficient and transparent manipulation of attribute hierarchies.

3.4 Preliminaries

In this section, we present details of the ROLAP server architecture used to support both hierarchical and non-hierarchical queries. We also describe the distributed data cube query engine based on the RCUBE data cube model. Finally, we introduce some notations for the presentation of our hierarchal data structures (hMap, xMap, and nMap) used to support hierarchical queries.

3.4.1 ROLAP Architecture

Contemporary data warehouses have grown enormously in size, with the largest now growing into the multi-terabyte range. For these massive data sets, multi-CPU systems offer great potential. The Sidera server was designed from the ground up as a high performance ROLAP indexing and query engine. The cube generation algorithms are fully parallelized and are load balanced and communication efficient on both shared disk and shard nothing cluster architectures. Methods for both full cube (all 2^d views) and partial cube ($< 2^d$) materialization are supported [6, 24, 25].

Explicit multi-dimensional indexing is provided by a forest of parallelized r-trees. The r-tree [12] indexes are packed using a Hilbert space filling curve [29] so that arbitrary k-attribute range queries more closely map to the physical ordering of records on disk. The Hilbert-ordered records are striped across each of the p disks of the parallel machine where each striped partition forms a partial r-tree index.

Queries are distributed to each of the p nodes in parallel, allowing each of the processing nodes to participate equally in the resolution of every query. Load balancing errors due to set partitioning are typically less than 2%. In effect, each node

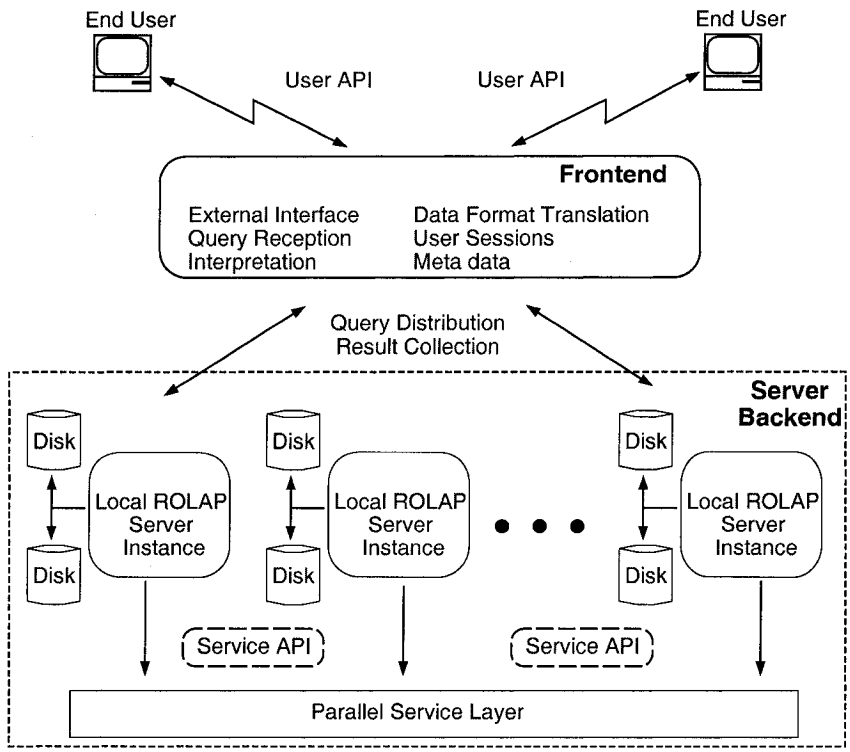


Figure 3.1: The Parallel ROLAP Server Architecture.

serves as an independent ROLAP server, requiring no direct knowledge of peer processing nodes. Figure 3.1 provides a simple illustration of the hardware/ software architecture for the Sidera query engine. Note that a Parallel Service API provides functionality (sorting, aggregation, communication, etc.) that allows local servers to operate independently.

3.4.2 The Query Engine Model

Before presenting the algorithms for hierarchical query resolution, we briefly discuss the core mechanisms for the original query engine. Algorithm 1 provides a high level description of Sidera's query resolution logic. The query interface is designed to be transparent, so that the user need not be aware of physical storage properties. We

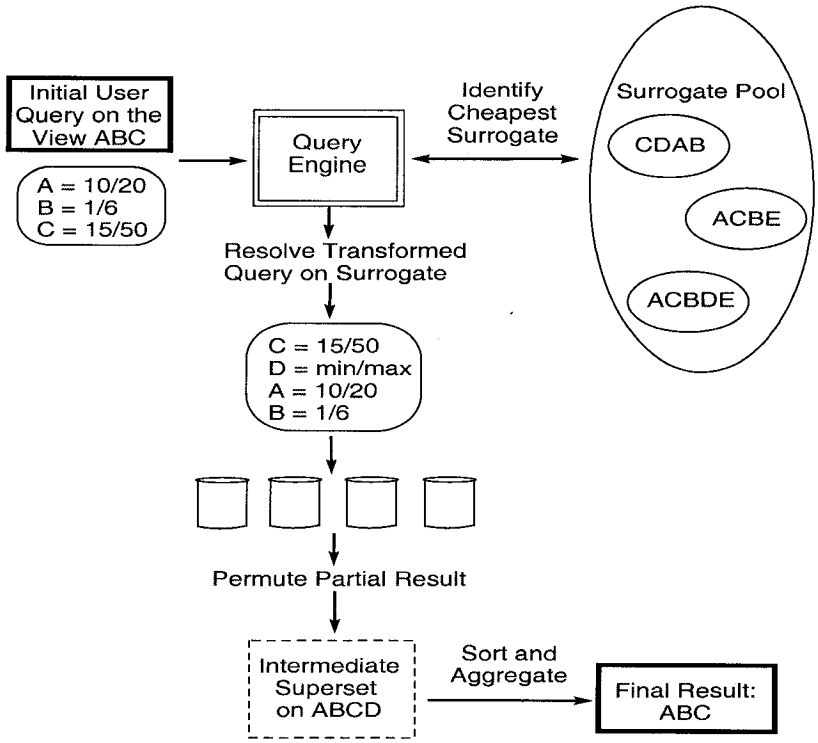


Figure 3.2: Basic query resolution, including surrogate exploitation.

refer to this model as the *virtual data cube*. Queries are passed to each node so that a partial result can be computed. Because partial data cubes are often constructed in practice, the system may need to identify *surrogates* since the user-specified view may not physically exist. Differing sort orders may also need to be addressed. Queries are transformed appropriately and partial results are obtained. A highly optimized Parallel Sample Sort [31] forms the basis of the aggregation, merging, and ordering operations. Figure 3.2 provides a graphical illustration of the Sidera query resolution model. Note how the end result exactly matches the user query, regardless of internal data characteristics. It is this query model that will be extended to support the attribute hierarchies.

Algorithm 1 Outline of Distributed *Partial* RCUBE Query Resolution

Input: A set S' of indexed group-bys, striped evenly across p processors $P_1, \ldots P_p$, and a multi-dimensional query Q.

Output: Query result deposited on front-end or distributed across the p processors.

1: Pass query Q to each of the p processors.
2: **if** the attributes in Q match those of disk view T **then**
3: select T as the *resolution target*
4: **else**
5: Locate *surrogate* group-bys T containing a superset of the attributes in Q. Select the one with smallest size as the resolution target.
6: **end if**
7: Transform Q into $Q\prime$ as per attribute ordering of T
8: Add *wildcard* values for the *peripheral* attributes.
9: In parallel, each processor P_j retrieves records R_j matching $Q\prime$ for its local data and reorders the values of R_j to match the order of Q. P_j also removes the redundant values for the peripheral attributes of T.
10: Perform a Parallel Sample Sort of $R_1 \cup R_2 \cup \ldots \cup R_p$ with respect to the attribute ordering of Q. While performing the sort, aggregate duplicate records introduced by the *peripheral* attributes of T.
11: **if** the query result is to be deposited on the front-end **then**
12: collect result via a MPI_AllGather (p node transfer).
13: **end if**

3.4.3 Hierarchical Attribute Representation

As noted, a dimension will often contain a hierarchy that represents a set of unique aggregation granularities on a given attribute. We introduce the following notation to support the presentation of our hierarchy data structures (hMap, xMap, and nMap). A hierarchy is constructed on top of a *base* attribute $A_{(1)}$ (i.e., the leaf), which can be interpreted as the finest level of granularity on that dimension. With our earlier example in Figure 1.1, the base attribute would be Product Number. The secondary attribute $A_{(2)}$ would be Product Type, while the tertiary attribute $A_{(3)}$ would be Product Category. Collectively, we refer to hierarchy levels above the base as sub-attributes. For a hierarchical attribute A, information captured by the attribute $A_{(i)}$ can always be obtained from $A_{(j)}$ when $i > j \geq 1$. This understanding is fundamental to the models presented in our research, in that data is physically stored only for the base attribute. As we will see, queries on other sub-attributes are mapped to this granularity level. We note at this point that while we retain only the most detailed data, the Sidera server stores a large number of pre-computed aggregates. As a result, the "most detailed level" may in fact refer to a cuboid that has already been heavily summarized.

We now describe the notion of hierarchy *linearity* [36, 21]. First, note that $A_{(i)}$ is considered a *direct descendant* of $A_{(j)}$ if $A_{(i)}$ is the child of $A_{(j)}$ in the hierarchy. A hierarchy is linear if for all direct descendants $A_{(j)}$ of $A_{(i)}$ there are $|A_{(j)}| + 1$ values, $x_1 < x_2 \ldots < x_{|A_{(j)}|}$ in the range $1 \ldots |A_{(i)}|$ such that

$$A_{(j)}[k] = \sum_{l=x_k}^{x_{k+1}} A_{(i)}[l]$$

where the array index notation [ ] indicates a specific value within a given hierarchy

level. Informally, we can say that if a hierarchy is linear, there is a contiguous range of values $R_{(j)}$ on $A_{(j)}$ that may be aggregated into a contiguous range $R_{(i)}$ on $A_{(i)}$. As a concrete example, the Time hierarchy is linear in that a contiguous range of *day* values — say, 15 to 41 — can always be aggregated into a contiguous range of *month* values — in this case 1 to 2. Note that our concept of linearity is directly related to the notion of summarizability [20], and is crucial in this context since non-linear data would effectively be non summarizable.

3.5 hMap: A ROLAP Hierarchy Data Structure

We begin the discussion of our new work by looking at the simplest case of strict, symmetric hierarchies. While such hierarchies can, in fact, be computed by other means, an understanding of our model for this simple case is fundamental to the discussion of the more complex unbalanced and non-strict hierarchies.

3.5.1 Preparing Symmetric Strict Hierarchies

We have described the different kinds of hierarchies in section 2.5. The hMap is used to model the simplest hierarchical forms. The Time hierarchy shown in figure 3.3 is what we refer to as an *implicit hierarchy*, one whose linearity should be self evident. The linearity of other attributes is not always immediately obvious. With an alphanumeric Product Number, for example, it is not even clear how a Product Number such as "BY26T7999" compares to one like "GT45J7586" (in terms of $<$ *or* $>$ operations). Therefore, the process of mapping ranges of Product Category or Product Type sub-attribute values to a corresponding range of Product Number values is not clearly defined.

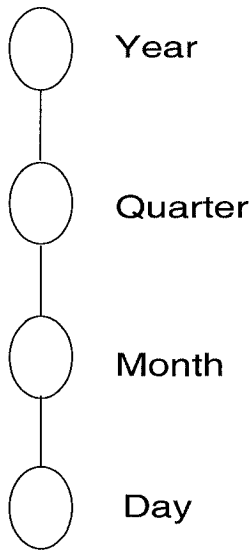


Figure 3.3: The Time hierarchy

Note that we cannot simply make a linear pass through the native data set and assign identifiers to records simply based upon the order in which they appear, as might be the case in a typical data warehouse. Hierarchical attributes mapped in this manner would be non-linear since an arbitrary mapping at the level of the base attribute would lead to non-contiguous ranges of non-base attributes. Instead, we enforce linearity by building *mapping tables* that are ordered by dimensions $A_{(k)} \times A_{(k-1)} \ldots A_{(1)}$. Figure 3.4 illustrates the mapping mechanism for a three-level Product hierarchy — Product Number (base), Product Type (secondary), and Product Category (tertiary). The mapping table consists of a set of n records, with n equivalent to the cardinality C of the primary attribute $A_{(1)}$ (i.e., Product Number). That is, for each product number, we create a record containing the Product Number and the corresponding Type and Category. A k-dimensional sort — with primary attribute Category, secondary attribute Type, and tertiary attribute Number — is performed on the n records.

Category	ID	Type	ID	Product	ID
Automotive	1	Brakes	1	XG27	1
				XY53	2
		Engine	2	GL75	3
				RT57	4
				RT91	5
		Interior	3	HJ45	6
				HY35	7
Household	2	Appliances	4	HK46	8
				UJ67	9
		Furniture	5	JW30	10
				NH22	11

Figure 3.4: The mapping for a simple three level Product hierarchy.

Upon completion, we associate the distinct values of each column with consecutive integer IDs.

3.5.2 hMap Data Structure

The mapping mechanism creates a linear hierarchy on a multi-level symmetric strict dimension (see figure 3.4). The mapping table is physically stored on the disk. In order to be used by the query engine, the table must be translated into an efficient in-memory data structure. In particular, the data structure must allow efficient mapping operations between hierarchy levels. This structure must support the following two basic operations: (i) Translation from a base level attribute value $A_{i(1)}$ to the corresponding sub-attribute $A_{i(j)}, j > 1$;(ii) translation from a sub-attribute $A_{i(j)}$ to the corresponding *range* on the base attribute $A_{i(1)}$.

The translation is accomplished with the multi-dimensional hMap data structure illustrated in Figure 3.5. The hMap is a simple data structure that consists of a three

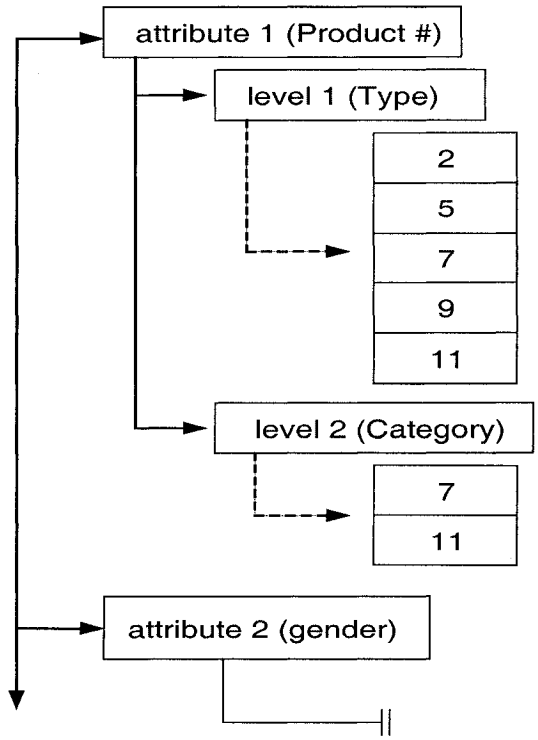


Figure 3.5: The hMap data structure, again using the Product hierarchy as an example.

level vector:

1. At the first level are the dimensions.

2. The second level contains the hierarchy levels for that dimension.

3. The third level contains the base-level partitioning values for the relevant hierarchy level.

Each core attribute A_i in the d-dimensional problem space is associated with $h_{A_i} - 1$ hierarchy maps, where h is the number of hierarchy levels for attribute A_i. No hierarchy map is associated with the base level of any hierarchy; these mappings are obtained indirectly. For a given level $A_{i(j)}, j > 1$, the associated map is made up

of the maximum value from the range on $A_{i(1)}$ corresponding to the current value of $A_{i(j)}$. We use Figure 3.4 as an example. Type 2 (Engine) corresponds to the base level (Product ID) range $3 \mapsto 5$.The second cell of the Type map therefore contains the value 5.

3.5.3 hMap Analysis

Because of the significance of hierarchy mapping within the query resolution model, hMap access time is of primary importance. In this regard, we note that the worst case query time is bounded as $O(\log |l_{d(A_i)}|)$, where $|l_{d(A_i)}|$ is the cardinality of the destination level of the hierarchy on A_i. To see why this is the case, consider the following. To map from a sub-attribute $A_{i(j)}$ to the corresponding *range* on the base attribute $A_{i(1)}$, we simply index directly into the hMap using the value of $A_{i(j)}$ as the map index t. The contents of the associated cell represents the maximum range value, while $map[t-1]+1$ is the minimum value. This operation can be performed in O(1) time.

By contrast, to map from a base level attribute value ε on $A_{i(1)}$ to the corresponding sub-attribute $A_{i(j)}, j > 1$, we must find the index position t such that $map[t] >= \varepsilon$ and $map[t-1] < \varepsilon$. Because the values of the map are sorted in ascending order, the query effectively reduces to a binary search on the destination map. The size of this map is $|l_{d(A_i)}|$. We therefore have a bound of $O(\log |l_{d(A_i)}|)$. Note as well that a mapping between arbitrary levels in the hMap can be represented as an O(1) mapping to the base level, followed by a mapping to any non-base level.

A second consideration for the hMap is its memory requirements, since we would like to save the bulk of these resources for buffering and query caching. Note that while we could guarantee O(1) for all operations on the hMap by including a base

level map, the cardinality of the base level can be quite large. There might be, for example, a million or more Products. By eliminating the base, the collective size of a d-attribute hMap using non-base levels exclusively is just:

$$\sum_{i=1}^{d} \sum_{j=2}^{h_{A_i}} |l_{j(A_i)}|$$

where h_{A_i} is the number of levels in the hierarchy for attribute A_i and $|l_{j(A_i)}|$ is the cardinality of level j for the hierarchy on attribute A_i. In practice, this would likely be no more than a few dozen kilobytes for large data cube problems.

3.6 xMap: Another ROLAP Hierarchy Data Structure

While the hMap performs very well in the limited context of simple symmetric hierarchies, it is not sufficient in its current form to support more sophisticated hierarchical representations. In this section, we describe an extended form of the hMap, called the xMap, which can in fact support strict, ragged hierarchies.

3.6.1 Extending the hMap

The framework described in the previous section demonstrates how the hMap data structure can be used to transparently translate user queries between arbitrary levels of a symmetric strict dimension hierarchy. As noted, however, simple hierarchies such as the one used in the running example can be modeled relatively easily using the standard Star Schema (though much less effciently). We can collapse the logical hierarchy tables into a (slightly redundant) single table. During a typical query, the user can issue a selection constraint that identifies the relevant values on a specific

hierarchy level column. A simple GROUP BY operation can then be performed, rolling up hierarchical aggregates at the appropriate level of granularity.

In this context, the hMap offers what might be considered minor advantages. For example, for large dimensions like Product and Customer, the hMap may significantly reduce storage requirements, especially for deep dimension symmetric strict hierarchies with modest member counts at each hierarchy level (i.e., a fairly common scenario). Moreover, the hMap can be used for simple hierarchy queries —those not requiring constraints on peripheral Dimension attributes— without requiring a join operation with the Dimension tables (where the hierarchy values would normally be stored).

Nevertheless, to be truly useful, mapping models like the hMap must be capable of handling more than just simple symmetric hierarchies. Perhaps the most common such case is the ragged strict hierarchy (discussed in section 2.5). Figure 3.6 provides a simple illustration of a typical ragged strict Product hierarchy (often referred to as unbalanced). Note that a given path may have an arbitrary level of nodes before finally reaching the leaves.

In terms of our mapping framework, our goal then is to extend the simple model of the hMap so that it is also able to support ragged hierarchies without sacrificing performance or introducing unacceptable complexity. As with the hMap, we must maintain the notion of attribute linearity and, in addition, we must provide translations between arbitrary levels of the now possibly ragged hierarchy. So we want to modify the data structure but still provide the same functionality.

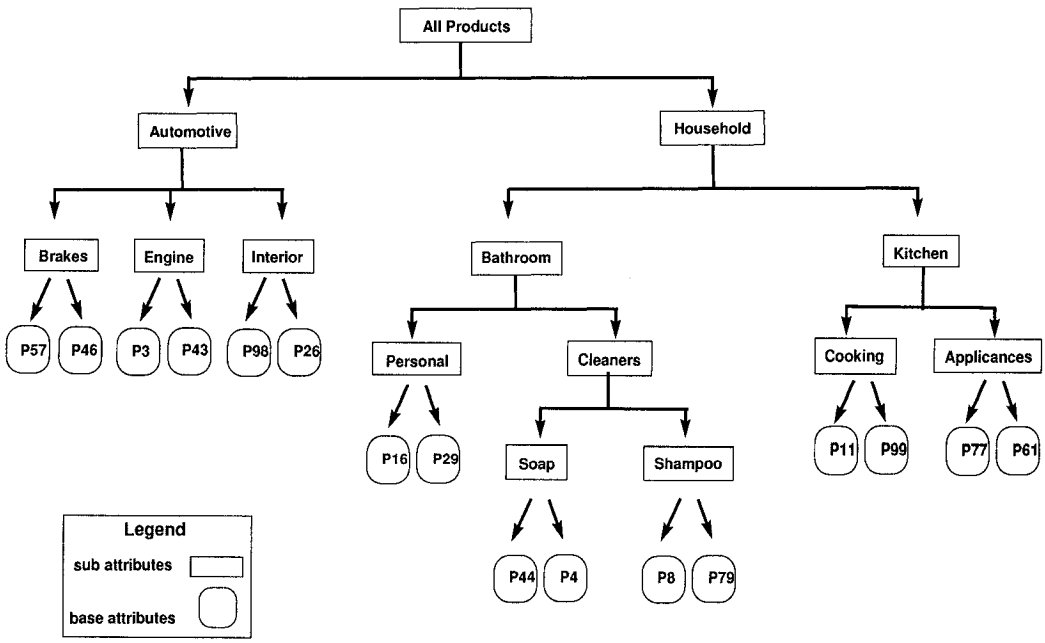


Figure 3.6: An unbalanced/Ragged Product hierarchy. The OLTP product numbers are prefaced with "P".

3.6.2 Preparing Ragged Strict Hierarchies

We use the same mechanism we used for the hMap to create the mapping table for the xMap. Again, we will use a mapping table to define the structure of the map itself. In this case, we begin by identifying the deepest level of the current hierarchy. Using Figure 3.6 as the running example, the maximum depth would be five. For all mapping records with less than five values, we then add dummy members for the intermediate missed values to fill out the unused fields. Dummy members ("-") are added to paths that do not descend to the bottom of the dimension hierarchy. A multi-dimensional sort is then performed to produce the table depicted in Figure 3.7. Note the following two characteristics of the new mapping Table:

1. The base attributes (i.e., product numbers), which are integer-based surrogate

Level 1	ID	Level 2	ID	Level 3	ID	Level 4	ID	Level 5	ID
Automotive	1	Brakes	1	---	---	---	---	P46	1
Automotive	1	Brakes	1	---	---	---	---	P57	2
Automotive	1	Engine	2	---	---	---	---	P3	3
Automotive	1	Engine	2	---	---	---	---	P43	4
Automotive	1	Interior	3	---	---	---	---	P26	5
Automotive	1	Interior	3	---	---	---	---	P98	6
Household	2	Bathroom	1	Personal	1	---	---	P16	7
Household	2	Bathroom	1	Personal	1	---	---	P29	8
Household	2	Bathroom	1	Cleaners	2	Soap	1	P4	9
Household	2	Bathroom	1	Cleaners	2	Soap	1	P44	10
Household	2	Bathroom	1	Cleaners	2	Shampoo	2	P8	11
Household	2	Bathroom	1	Cleaners	2	Shampoo	2	P79	12
Household	2	Kitchen	2	Cooking	1	---	---	P11	13
Household	2	Kitchen	2	Cooking	1	---	---	P99	14
Household	2	Kitchen	2	Appliances	2	---	---	P61	15
Household	2	Kitchen	2	Appliances	2	---	---	P77	16

Figure 3.7: The mapping table for a ragged product hierarchy.

keys, are consecutively numbered.

2. Level IDs do not cross level boundaries and may be repeated in the same column.

3.6.3 xMap Data Structure

Now that the mapping table has been constructed, we must use it to load the appropriate data structure. Note, however, that the three level vector of the hMap is no longer adequate. Instead, we propose the xMap. Figure 3.8 illustrates the xMap tree, using the data of the running example in figure 3.6. xMap essentially uses a b-tree style data structure, except that the nodes will be of arbitrary size. The tree is constructed so as to provide an efficient representation of the original ragged strict hierarchy. Like the hMap, it only stores base attribute delineation points. No

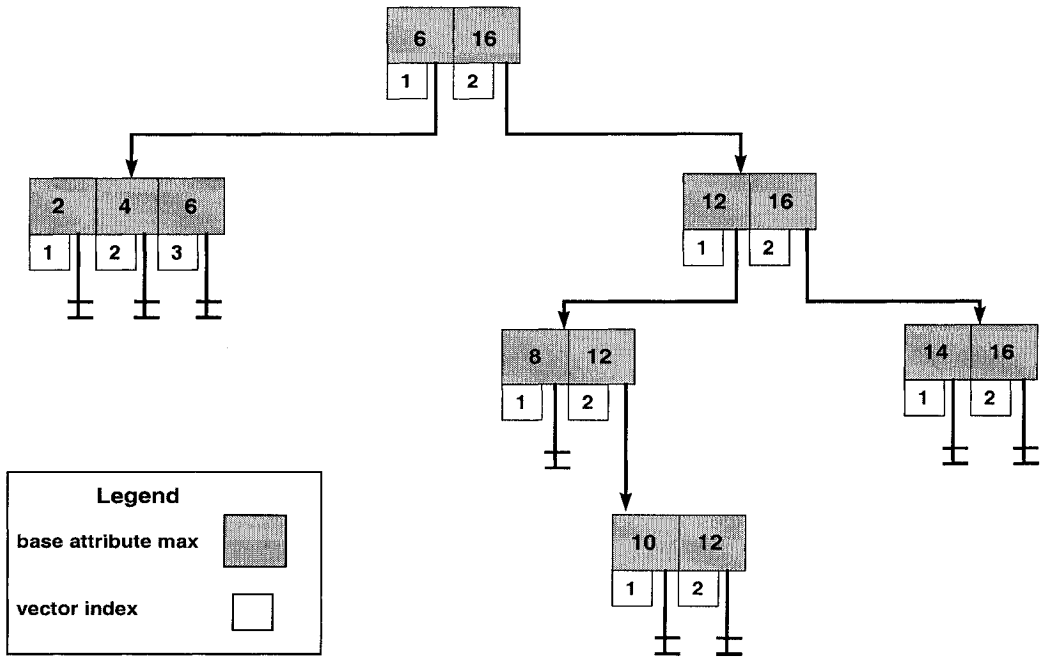


Figure 3.8: The xMap data structure.

additional data need be supplied. We note that query specification for ragged (unbalanced) hierarchies of arbitrary depth is somewhat more complex than would be the case for simple balanced hierarchies.

Specifically, it must be possible to define an arbitrary but exclusive path through the hierarchy tree. In order to translate between arbitrary levels of the ragged hierarchy, we must have a standardized way of accessing hierarchy levels. In this respect, we have decided to move towards an MDX-style syntax for our queries. For example, a hierarchy member from our running example might be identified as Product. [Household]. [Bathroom]. [Personnel]. [Children]. Here, the trailing "Children" is a language keyword identifying all descendants of the Personnel member.

Because the xMap contains only integers, text values will be mapped to integers using a simple hashing mechanism. Basically, hashing a sub-attribute's text value will

return the number defined in the mapping table. So, for example, Bathroom would be 1, while Shampoo would be 2. We can combine the ID values to create a unique navigational pathway. Using the same example, Household → Bathroom would be 2 → 1, while Household → Bathroom → Cleaners → Shampoo would be 2 → 1 → 2 → 2. Note the following with respect to xMap navigation:

1. Duplicate ID values in the same column are disambiguated by the path prefix.
2. The values in the path correspond directly to the index values in the xMap.

Recall that our primary objective for the xMap data structure is to provide efficient translation between the base attribute and sub-attributes for the ragged hierarchies. For the MDX query syntax, this bi-directional mapping is accomplished as follows:

1. **Base to sub attributes**:
    - Simply traverse the tree to find the location of the base attribute
    - As we traverse, we record the index values for the path we take
    - For example, given Product 9 (P4), we would generate the path 2 → 1 → 2 → 1
    - This result then allows us to translate the Product number to any level of the Hierarchy
        - Level 1: Household (2)
        - Level 2: Bathroom (1)
        - Level 3: Cleaners (2)
        - Level 4: Soap (1)

2. **Sub attributes to the base:**

   - Here, the MDX-style query is mapped to a navigational path.
   - This path takes us to the proper tree node
   - Since the base attributes values are stored within the node in sorted order, the range is simply calculated as the max/min values for the associated index position
   - For example, Household → Kitchen → Appliances would map to 2 → 2 → 2, giving us products 15 and 16

3.6.4 xMap Analysis

The formal characteristics of the xMap are similar, though not identical to those of the hMap. To map from a base level attribute value to the corresponding sub-attribute $A_{i(j)}, j \geq 1$, we simply traverse the tree until the appropriate leaf-level position is found. A sub-attribute string is constructed with no additional cost. Consequently, the worst case bound on this form of translation is simply $O|l_{max(A_i)}|$, effectively the longest path in the A_i hierarchy. While the xMap has an irregular fan out (i.e. a non uniform number of children), note that the number of levels in the tree is directly related to the number of levels in the hierarchy. Specifically $|l_{max(A_i)}| = \text{levelCount}(A_i)$ - 1.

When mapping from a sub-attribute to the base, we must first identify the level corresponding to the sub-attribute, and then perform the base attribute mapping on the appropriate tree node. Worst case mapping time is therefore $O|l_{max(A_i)}| + O(log|l_{d(A_i)}|)$. Here, $O|l_{max(A_i)}|$ is again the depth of the hierarchy on A_i while $|l_{d(A_i)}|$ is the cardinality of the destination level of the hierarchy A_i.

In terms of physical resources, we note that while the structure of the xMap is certainly different than that of the original hMap, its content is effectively the same. That is, it stores a compact representation of the non-base hierarchy levels for a given attribute. As such, the memory footprint for the xMap in an asymptotic sense is equivalent to that of the hMap. Finally, we note that xMap can handle both balanced and ragged strict hierarchies without any problem.

3.7 nMap: Non-Strict Data structure

3.7.1 Motivation for the nMap

In Chapter 2, we presented a summary of some of the more important hierarchy forms found in real world applications, including symmetric and ragged hierarchies. In the previous sections, we have presented data structures that allow us to model and query such hierarchies efficiently. As previously noted, however, in each of these cases we assume the existence of *strict* hierarchies that exclusively define one-to-many relationships between parent and child nodes in the data model. For situations in which this is not the case, however, the xMap data structure and processing model is not completely sufficient.

Nevertheless, such hierarchies can and do occur in practical applications and must be efficiently supported. Consequently, we must be able to extend the current framework without significantly compromising its most important and most powerful features. It should be clear that we cannot use the xMap model directly as this architecture relies upon the linearity of base level members, which may not be the case when child members are associated with multiple parent nodes.

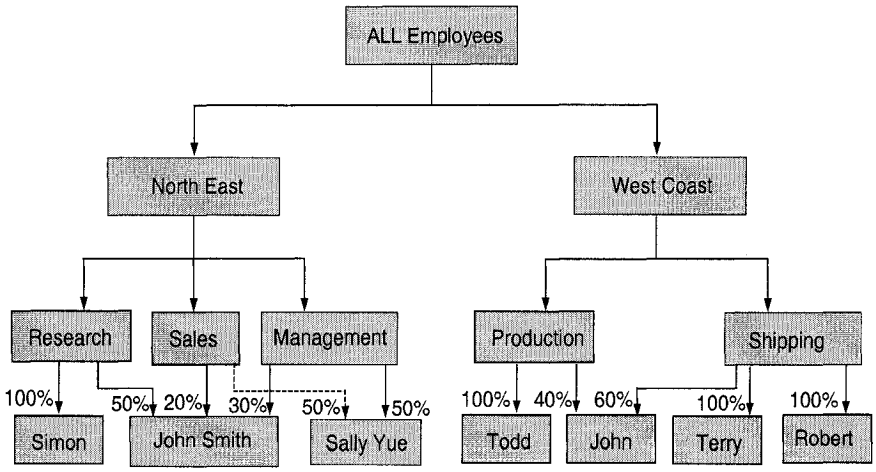


Figure 3.9: A simple non-strict hierarchy.

Figure 3.9 provides an illustration of a small but relatively typical non-strict hierarchy. In this case, we can see a many-to-many relationship between the leaf level and its immediate parent. Note as well that strict one-to-many relationships are found in the remaining levels. In fact, a mixed model such as this would be quite common. With respect to the non-strict relationships, we can see that the relevant edges have been augmented with numbers representing the distribution of resources for a given child. For example, the employee "John Smith" is associated with three parents, indicating that he is a member of the Research, Sales, and Management departments. As such, his contribution to the cube measure relevant to this environment must be shared across all three divisions. In this case, the cube designers/builders have determined (in consultation with users) that an appropriate distribution would be 50%, 20%, and 30% respectively. It should be clear that the relationship between the Department and Employee levels is many-to-many and that the task of disambiguating query paths is now more difficult and possibly less efficient.

One approach to resolving many-to-many range queries would be to utilize an

existing multi-dimensional query index such as an r-tree, grid-file, etc, in order to accurately map parent nodes to relevant child values. However, while such an approach would in fact work, doing so introduces considerable complexity into the existing model. Moreover, the number of pointers associated with such a model grows considerably as the hierarchy becomes more complex. Given that our primary design goals are (a) simplicity (b) minimal memory requirements (c) fast lookup times, we would prefer to have a more appropriate mechanism to support non-strict dimension hierarchies.

3.7.2 Preparing Non-strict Data

Before we present the details of the nMap data structure, we must first discuss the preparation of the data that will eventually be included in the nMap. Recall that the hMap and xMap utilized the notion of mapping tables to ensure attribute linearity. While non-strict hierarchies do not of course support the same strict linear ordering, it is in fact possible to benefit from a pre-ordering of the data. Recall that our fact table will again represent data at the most detailed storage level. With respect to Figure 3.9, this suggests that we will be storing measure information at the level of individual employees. In a Rollup operation, this implies that measurement totals must be divided between nodes at the parent level. This distribution information must be physically stored in order to permit this operation.

Though we know of no current OLAP application that directly targets many-to-many queries, one can "manually" support these operations at the table level using a technique that employs what are known as *Bridging Tables* [19, 18]. Here, we create a relational table that physically stores the measure distributions. In turn, this table can be accessed during query resolution in order to "match" parents to children. It is

ID	Text Values	Descriptive Fields
1	Simon	...
2	John Smith	...
3	Sally Yue	...
4	Todd	...
5	John	...
6	Terry	...
7	Robert	...

Table 3.1: The base level table

important to note, however, that the use of the bridging table significantly complicates the creation of queries and, as we will see in the Chapter 4 the use of additional joins creates serious performance problems.

Our objective therefore is to build upon the concept of the bridge table so as to extend the current model to allow it to efficiently and transparently support many-to-many hierarchical relationships. To begin, we review the tables that would be representative of the bridge table architecture.

1. **The Base Level Table** (Table 3.1): The base table simply contains the values of the leaf members and the associated integer ID (plus the remaining dimension attributes).

2. **The One-to-Many Mapping Table** (Table 3.2): Strictly speaking, this is not part of the many-to-many mapping. However, is important to illustrate the fact that resolution of the leaf level query would "pass through" this level before reaching the bridge table.

3. **The Bridge Table** (Table 3.3): This table stores three values per row and can be joined to either Table 3.1 or Table 3.2 via its first or second columns.

Level 1	ID	Level 2	ID
North East	1	Research	1
North East	1	Sales	2
North East	1	Management	3
West Coast	2	Production	4
West Coast	2	Shipping	5

Table 3.2: The mapping table for one-to-many relationships.

Base Level	Level 2, ID	%Distribution
1	1	100
2	1	50
2	2	20
2	3	30
3	2	50
3	3	50
4	4	100
5	4	40
5	5	60
6	5	100
7	5	100

Table 3.3: A simple bridging table

3.7.3 The nMap Data Structure

Now that we have described the bridging model that is used to store the definitions for the many-to-many relationships, we turn our attention to the in-memory representation of this information. In particular, our objective is to provide rapid *two-way* translation between level k and level $k-1$ in the dimension hierarchy. To do so, we will use a complementary pair of lists, $nmap_k$ and $nmap_{k-1}$. Simply put, $nmap_k$ provides translation to level $k-1$, while $nmap_{k-1}$ provides translations in the opposite direction.

Figure 3.10 illustrates the nMap structure for our running example. Each individual list in the map(s) consists of value pairs $< ID, R >$, where ID corresponds to the ID of the translated level and R refers to the associated distribution ratio. As a concrete example, let us examine the first list in Figure 3.10(a), the translation map for level k. This can be interpreted as follows. The list itself represents the map associated with ID 1 at level k, in this case the Research Department. The associated value pairs are $< 1, 100 >, < 2, 50 >$. This tells us that the Research Department contains two employees whose ID values at level $k-1$ are 1 (Simon) and 2 (John). Moreover, we know that 100% of Simon's contribution to the cube measure is associated with this department, while just 50% of John Smith's contribution is relevant to Research.

In contrast, let us examine $nmap_{k-1}[5]$ to see how the second map is utilized. Here, the fifth list is associated with ID 5 at the leaf level, in this case employee John. For this employee, we learn that he is a member of two departments, 4 (Production) and 5(Shipping). In turn, his measurement values should be divided so as to link 40% to the first department and 60% to the latter.

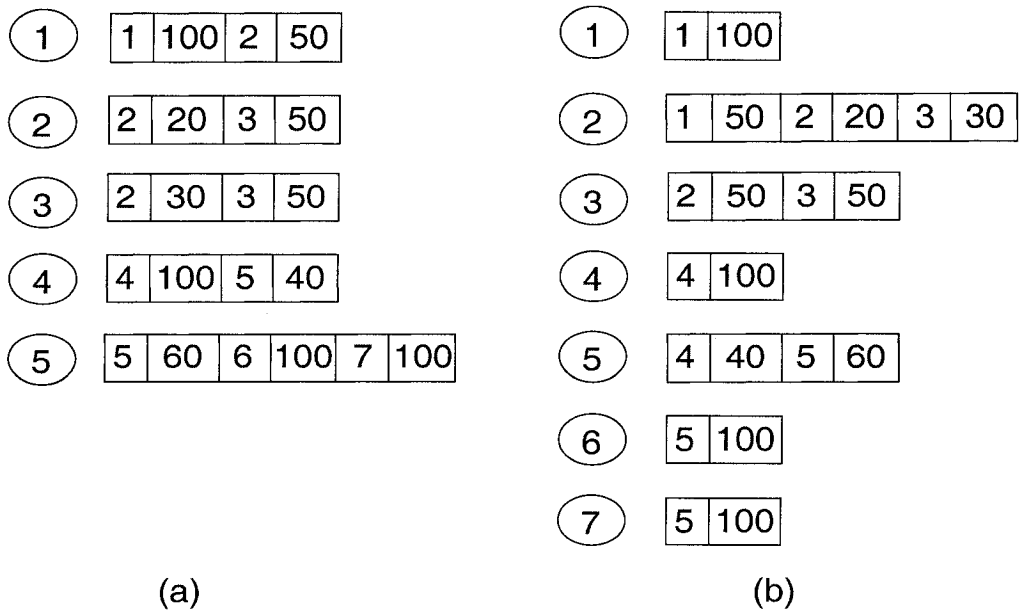


Figure 3.10: The nMap data structure. (a) Level 2 to Level 3 (Base Level). (b) Base Level to Level 2.

In terms of the physical structure, note that the mapping lists are physically represented as arrays; no pointers are required. Further, both $nmap_k$ and $nmap_{k-1}$ are defined as *arrays of arrays*, thereby allowing the internal lists to be of arbitrary size, as would be required in a practical application. Finally, note that because the construction and loading of the maps is based upon the underlying ID maps of the bridge table model, the ID values of the nMap are always listed in sorted order.

While the nMaps are extremely compact, it should be clear that the ratios themselves are redundantly shared between $nmap_k$ and $nmap_{k-1}$. Moreover, while our running example shows the ratios in integer form (the most likely case in practice), it would be possible to define floating point versions of these values. As a result, it is of prime importance to minimize the storage requirements for the ratios. The

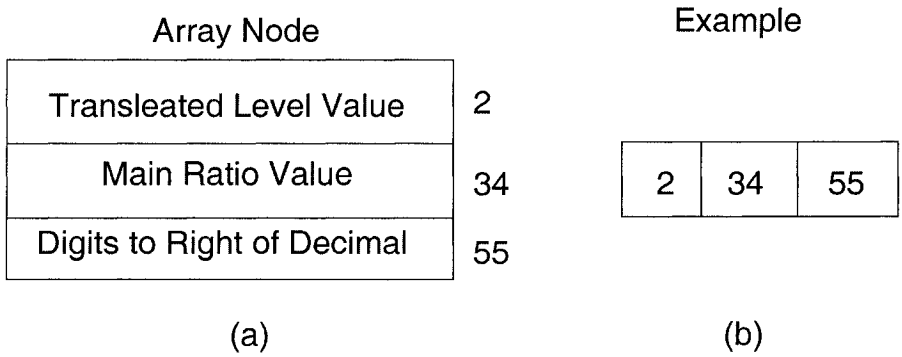


Figure 3.11: The nMap Array Node for ID = 2, ratio = 34.55.

solution is relatively straightforward. Note that in the more complex case of floating point representation, we can reduce the ratio to a $< integer, decimal >$ format (e.g., 34.50 or 28.95). Clearly, the first component is limited by the interval $(0, 100]$. Now, while the size of the second component is unbounded in theory, a practical limit is likely to be two decimal places of precision. This is the case since the value represents a user defined estimate of the distribution ratio; therefore, precision beyond two decimal places would likely not be meaningful. As such, we can represent the range by the interval $[0, 99)$. Given the above restrictions, we define the distribution value as consisting of the triple $< ID, ratio_{prefix}, ratio_{suffix} >$. While the size of the ID may be adjusted to suit the ID range, the *ratio pair* need only occupy $\lceil$ 7 bits + 7 bits $\rceil$ = 2 bytes of storage, significantly reducing the impact of ratio redundancy. Figure 3.11 provides a simple example. In practice, we would store node triples as simple C *structs*.

3.7.4 Query resolution in Mixed Mode environments

Because one-to-many and many-to-many hierarchies must co-exist within the same dimension hierarchy, it is important to illustrate the query processing logic within a

mixed mode xMap/nMap environment. In Figure 3.12, we see a relatively complete representation of our running example. To avoid any confusion, we will refer to Region as Level 1, Department as Level 2, and Employee as Level 3. In addition to the mapping structures defined in Figure 3.10, we now add the xMap component for Level 1 and Level 2.

An important point with respect to the mixed mode mapping is the effect upon linearity. As previously noted, the xMap (and the simpler hMap) relies upon a linear ordering of base level attributes. In turn, this ordering supports the efficient traversal of the xMap structure. When non-strict hierarchies are utilized, the existence of multi-parent relationships prevents the definition of a unique linearization. In other words, a child can be linked to a set of non-contiguous parents. Nevertheless, in order to integrate the xMap and the nMap, linearity must be maintained for the xMap components. We ensure this as follows. For an xMap between level k and $k+j$, the level $k+j$ becomes the *logical base*. For all *consecutive* levels above $k+j$, the xMap stores the maximum values for the associated logical level. This process continues until either (i) we reach the root or (ii) another xMap level is encountered. Consequently, the xMap maintains its unique linearity property while, as we will see in the description that follows, allowing a clean integration with the nMap.

Recall that our objective is to provide bi-directional translation between base and sub-attributes for both strict and non-strict hierarchies. For the purposes of illustration, we will assume that the nMap is housed at the base level though, as noted above, this is not required. Assuming the standard MDX-style query language, we may resolve queries as follows:

1. Base to sub-attributes.

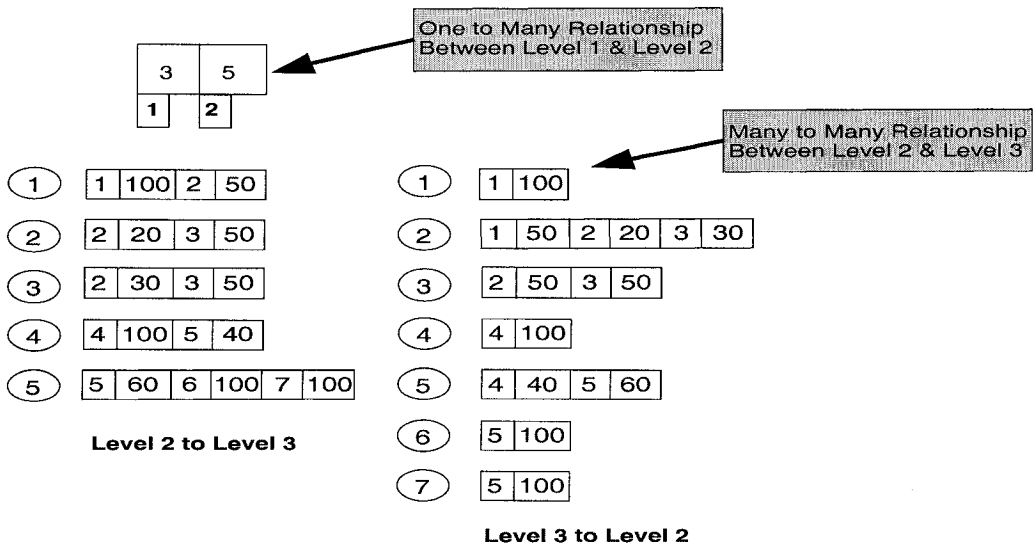


Figure 3.12: A more complete example showing the relationship between the nMap and the xMap.

- Let us assume that we have been asked for the measure value corresponding to Sally Yue (Level = 3, ID = 3) and, further, that this result should be "rolled up" to some arbitrary level of the hierarchy.
- Unlike the simple xMap, we cannot begin with a tree traversal since the member Sally Yue may have non-contiguous parents in the tree above.
- We therefore begin by reading the nMap array corresponding to Sally Yue (array index 3 in the second nMap structure). This tells us that ID 3 is associated with members 2 and 3 at the logical base defined at level 2. In addition, ID 3 contributes 50% for members 2 and 50% for members 3 at level 2.
- We now descend from the root. Since both logical base values are ≤ 3, we need only descend a single path of the xMap (i.e., the left path of level

1). As we have done with the xMap, we record the level values that define the paths to the (true) base level. As we pass through the nMap level, we augment the parent nodes with the relevant distribution ratios.

- In our current example, the traversal would produce the following results:
  - Level 1: North East (1)
  - Level 2: Sales ($< 2, 50\% >$) and Management ($< 3, 50\% >$)
- At level 2, we see that the total measure value for Sally Yue should be split equally between the Sales and Management departments.

2. Sub-attributes to Base.

- The translation of sub-attributes to base is actually more straightforward. We will assume that we have been asked for a drill down operation on the West Coast region.
- We begin by mapping the query to a navigational path. This path brings us to the appropriate tree node in the logical base. From here, we can read the appropriate arrays from the nMap.
- Since the values at the logical base are stored in sorted order, the range can simply be calculated as the max/min values for the associated index(es).
- In the current example, West Coast would map to ID = 2 at level 2, giving us two departments, Production (ID = 4) and Shipping (ID = 5). We then use the nMap to obtain the base levels values, as well as the related ratios. The resulting base values for West Coast are as follows:
  - Production (4): Todd $< 4, 100\% >$, John $< 5, 40\% >$

– Shipping (5): John $< 5, 60\% >$, Terry $< 6, 100\% >$, Robert $< 7, 100\% >$

3.7.5 nMap Analysis

As is the case for the hMap and the xMap, performance is of primary concern during the translation operations. In terms of the mapping from base to sub-attribute $A_{i(j)}, j > 1$, we note the following. The first step in the process is the selection of the relevant array in the nMap data structure. Since the arrays are indexed by their predefined mapping values, this operation can be performed in O(1) time. Processing of the array itself is $O(k)$, where k is equivalent to the parent count in the many-to-many mapping. In practice, however, the value of k is quite small in the OLAP content, typically < 5. For each path $j, j \in k$ so defined, we traverse the mixed mode tree in the standard fashion (i.e., using the xMap). Consequently, the base to sub-attribute access cost can be bounded as $O(1) + O(k) + O(k * (levelCount_{A_i} - 1)) = O(k * (levelCount_{A_i} - 1))$. Here, $levelCount_{A_i}$ is the number of levels between the consecutive one-to-many relation-ships in hierarchy A_i. Again, we note that the value of k is quite small so that the practical increase in processing cost is a small factor above that of the standard xMap.

With respect to the translation from sub-attributes to base, we begin by descending the xMap in the standard fashion to obtain the max/min range on the associated nMap. This is a simple $O(levelCount_{A_i} - 1)$ operation. We must now process the $O|l_{d(A_i)}|$ members we find there, where $|l_{d(A_i)}|$ refers to the cardinality of the destination level. For each of the members of this set, we record their k parent/ID pairings. The worst case cost of the operation is therefore $O(levelCount_{A_i} - 1) + O(k * |l_{d(A_i)}|)$. In practice the second component will dominate and will likely create a larger cost than the base to sub-attribute mapping. We note the following, however. First, the

practical value of k is quite small, while $|l_{d(A_i)})|$ overestimates the actual cost since only a small portion of a given level is likely to be delineated by a given max/min pair. Second, the processing of the parent/ID mappings is a minimal cost for any many-to-many hierarchy algorithm since all relevant ratios must be processed. Given that, the nMap provides its result with very little additional overhead.

In terms of the memory requirements, we note the nMap is designed to minimize the storage requirements of the bridge table data. It represents all relevant information without the use of pointers and further minimizes ratio values with a compact two-byte format. In a mixed mode format, integration of the nMap has no impact upon the storage required for the associated xMap.

3.8 Caching and Software Model

In practical OLAP environments, query processing is typically improved through the use of a cache manager that records the results of previous user queries. We have augmented the hierarchy processing subsystem with a simple caching model that serves as a *proof of concept* for a more comprehensive system. In this section, we discuss the caching framework used to support hierarchical ROLAP queries, as well as the software architecture utilized on each processing node.

3.8.1 Caching Hierarchical ROLAP Queries

While the parallel indexing facilities provided by the Sidera server support effective disk-to-memory transfer characteristics, optimal query response time relies to a great extent on an effective caching framework. Given the sizable memory capacity of our parallel ROLAP server, it is expected that a significant proportion of user queries will

be answered in whole or in part from a *hot cache*. Sidera provides a natively multi-dimensional, hierarchy-aware caching model. Specifically, resolved partial queries are cached on each node of the parallel machine. For a new k-attribute range query, with ranges $\{R_1, R_2, \ldots, R_k\}$, the cache mechanism must determine if, for each attribute A_i, the range R_i of the user query is a subset of the range on A_i of the cached query. If, for all k attributes, subset ranges are found, the cached query is used in place of disk retrieval. At present, the Cache Manager does not process partial matches. That is, it does not answer queries partly from the cache and partly from disk. This, however, is the subject of ongoing research. Specifically, the logic required for this form of decision making is being integrated into the query optimizer.

With respect to hierarchies, *metadata* is maintained by the Cache Manager and is used in conjunction with the hMap, xMap, and nMap to perform translations between hierarchy levels. For a k-attribute user query, an arbitrary number of attributes can be re-mapped simultaneously. Note that queries are cached in their preliminary state — that is, they are cached in their base attribute form before final transformations have been applied. This permits hierarchies to be mapped to arbitrary levels — caching at levels above the base would prevent the cache from answering queries at finer levels of granularity. It is important to note that the cache forms the basis of the core *Five Form* query model. Specifically, all OLAP servers should be able to support at least five basic OLAP-specific queries: roll-up, drill-down, slice, dice, and pivot. The query engine transparently manipulates the cache contents to further refine previous user queries. A drill down, for example, is produced merely by translating hierarchy levels within the current cache.

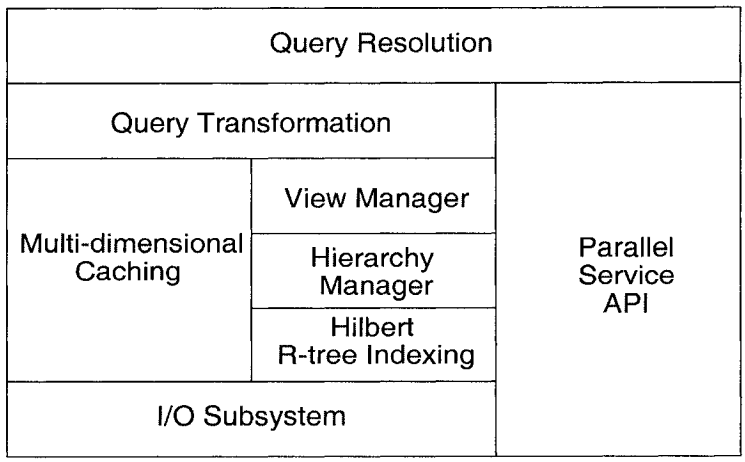


Figure 3.13: A block diagram of the module stack on each of the local processing nodes.

3.8.2 The Software Model

Taken collectively, the software architecture on each processing node forms a clean, modular design. Figure 3.13 illustrates how the hierarchy and caching components fit into the larger framework. In the current context, the primary modules are the Hierarchy Manager (i.e., hMap, xMap, and nMap), the Cache Manager, and the View Manager. The first two have already been discussed. The View Manager maintains meta data about the format and sort orders of views physically stored on disk. It is used when queries cannot be resolved from the Cache.

3.8.3 The Query Resolution Algorithms

The initial query engine, described in Section 3.4, was designed for simple, non-hierarchical attributes. With the addition of the hierarchy maps and the caching framework, the algorithms had to be extended to accommodate more complex queries. Algorithm 2 describes the updated algorithm for querying multi-dimensional data in

the presence of hierarchies. We stress that the core algorithm pre-dates the work in this thesis. Our purpose in this section is therefore to describe the integration of our mapping structures into the original model.

To begin, before query resolution actually takes place, the "raw" query is transformed, taking into account hierarchical specifications. An initial result is then obtained either from the cache or, if necessary, from disk. If obtained from the cache, the prepareCachedQuery function is used to re-order the cached attributes in the query buffer to match the order of the user query. Additional, non-specified attributes are dropped. If disk access is required, the initial data is retrieved via the r-tree indexes and is added to the cache. Query-specific post processing is then performed. Finally, because Sidera is a fully parallelized query engine, the partial results may need to be collected and merged before presentation to the end user. We use a standard gather operation from the MPI libraries (Message Passing Interface).

3.8.4 Query Transformations

Algorithm 2 utilizes a function called transformQuery to convert the user query into a hierarchy-aware form that can be utilized by the query engine. This algorithm is described in Algorithm 3. The primary function is to create new range and hierarchy arrays. The range array provides the new high/low values for each of the A_i attributes in the user query. These are specified in terms of the base attribute. The hierarchy array will continue to reflect the hierarchy level requested by the user but will be updated with wildcards to indicate full range matching on peripheral attributes. By peripheral, we mean dimensions that were not part of the user query but that may be part of the surrogate view selected to actually resolve the user query.

Algorithm 2 Query resolution in the presence of hierarchies

Input: A set M of user-defined query parameters, a hierarchy manager hM, a cache Manager cM, and a view manager vM.
Output: Fully resolved and concatenated query result.

```
1: load user query uQ with parameter set M
2: invoke transformQuery(uQ, hM, vM)
3: check the cache Manager for a match on uQ
4: if a valid match is found in the buffer then
5:    invoke prepareCachedQuery (cQ, uQ) to get the Initial Result I
6: else {otherwise, go to disk to answer the query}
7:    I = processQuery (uQ)
8:    add I to the cache
9: end if
   {do OLAP post processing}
10: final result R = postProcessing(uQ, hM, I)
11: if results required on front end then
12:    collect R with MPI_Allgather
13: end if
```

3.8.5 Post processing

Once the initial result set has been constructed in Algorithm 2, post processing must be performed in order to produce the final result. This process is described in Algorithm 4. Note that the post processing routines are completely oblivious to the source of the initial result (cache or disk).

The translateHierarchyValues() function is used to map base level values in the initial result set into their appropriate counterpart at the destination level of the hierarchy (as defined by the user query). The system uses the Hierarchy Manager (hMap, xMap, or nMap), and hierarchy array (constructed in Algorithm 3 for this purpose). A Parallel Sample Sort is performed to order records as per the user request and to permit efficient merging and aggregation. Note that the sorting subsystem is heavily optimized to minimize the movement of multi-value records. If surrogates

Algorithm 3 Query Transformation Algorithm

Input: A user-defined query uQ containing dimension set M, a hierarchy manager hM, and a view manager vM.

Output: Optimized query format.

```
1: retrieve the actual view V from the view Manager, where V contains dimension
   set T, M ⊆ T.
2: create a new attribute range array newR of size |T|.
3: create a new hierarchy range array newH of size |T|.
   {populate newR and newH}
4: for each attribute i in V do
5:   if uQ contains V[i] then
6:     low = the range minimum for V[i]
7:     high = the range maximum for V[i]
8:     l = hierarchy level for V[i]
9:     if l does not represent the base level then
10:      set newR.low = hM.getBaseLow(V[i], l, low)
11:      set newR.high = hM.getBaseHigh(V[i], l, high)
12:    end if
13:  else
14:    set high/low wildcards
15:  end if
16: end for
17: update the current user query uQ with newR, newH, and V.
```

or hierarchies have been specified, some form of additional aggregation will also be required. This is the purpose of the orderAndAggregate() function. At this point, the result is ready for its return to the user.

Algorithm 4 ROLAP Post Processing Algorithm

Input: user query uQ, initial result I, hierarchy manager hM
Output: final result R

```
1: set the user-specified view U from uQ
2: set the actual view V from uQ
3: if uQ contains hierarchies then
4:    invoke translateHierarchyValues(uQ, hM, I)
5: end if
6: do parallel sample sort
   {permute intermediate results as per user request}
7: if a surrogate was used or a hierarchy is required (or both) then
8:    R = orderAndAggregate(I);
9: else
10:   R = arrangeSortedRecords(I);
11: end if
12: return R;
```

3.9 Review of Research Objectives

In Section 3.3, we identified a number of objectives of our research. We now review those goals to confirm that they have in fact been accomplished.

1. **Build a simple design that is easy to implement and update.** We began by describing the hMap, a simple structure that consists of a three level vector. Unbalanced (and symmetric) hierarchies are supported by the xMap, using a b-tree style data structure (with arbitrary node fanout). Finally, the

nMap employs two primitive multi-level arrays to add many-to-many hierarchy support.

2. **Minimal memory requirements**. The cardinality of the base level can be quite large. There might be, for example, a million or more Products. For both symmetric and ragged hierarchies, we are able to minimize the impact of large cardinalities by constructing the hMap and xMap to store only base attribute *delineation* points. With many-to-many relationships, more extensive information may be required. Here, the nMap stores the bridge table data but minimizes memory consumption in three ways: no pointers are required, null values are not stored, and the duplicated ratios are stored as just two bytes.

3. **Maintain the notion of attribute linearity**. Linearity is directly supported in the hMap and xMap. In the more complex nMap, we are able to maintain the linearization concept with the introduction of *logical* base levels.

4. **Efficient translation between arbitrary levels of the hierarchy at query run-time**. The hMap and xMap structures are extremely fast, with access time bounded by the level count. The xMap can be slightly more expensive, due to the more complex many-to-many mappings, but the structure still reduces arbitrary translations to a small number of operations in practical environments.

5. **Support for multiple hierarchy forms**. In practice, out mapping methods can support Symmetric Strict Hierarchies, Symmetric Non Strict Hierarchies, Ragged Strict Hierarchies, and Ragged Non Strict Hierarchies.

6. **Provide a multidimensional caching framework to support fundamental query types.** We describe a mechanism to improve real time query performance through the use of a hierarchy-aware caching model.

3.10 Conclusions

In this chapter, we described three approaches (hMap, xMap, nMap) for modelling several important hierarchy forms found in the real world. Our models rely upon the construction of very efficient, in-memory data structures, each designed for a particular hierarchy form. That being said, we have also demonstrated how distinct mapping models are likely to be integrated in practical environments. The end result is a comprehensive framework that supports fast translation between arbitrary levels of dimension hierarchies at run-time. Moreover, we have described a caching subsystem that cleanly supports the full suite of mapping data structures.

As noted in Section 3.1, relatively little is published work in the area of hierarchy implementation for relational environments. Given the significance and size of the underlying problem, there would appear to be a genuine need for this type of research. In our case, we have contributed to the literature by providing a comprehensive framework for the implementation of hierarchy forms that are not easy or efficient to model with current techniques. Of particular significance is the fact that we have grounded our approach with extensive experimental evaluation. These results are the subject of chapter 4.

Chapter 4

Experimental Results

4.1 Introduction

In the previous chapter, we described a full suite of data structures and query methods in support of real world OLAP dimension hierarchies. In the current chapter, our focus turns to the effectiveness of these methods across a broad range of common input parameters. Specifically, we provide experimental results that assess the ability of the augmented query engine to efficiently support queries in the presence of hierarchies. Our primary concerns are the computational overhead introduced (or reduced) by the mapping data structures, as well as direct comparisons with alternate models.

The chapter is organized as follows. In Section 4.2, we discuss the test environment as it relates to the hardware, software, and data that we use in our evaluation. In Section 4.3, we will look at a sequence of tests in order to highlight the importance of our methods with respect to the resolution of hierarchical queries. Computational overhead will be discussed in Section 4.3.1, comparisons with existing techniques in Section 4.3.2, Section 4.3.3, and Section 4.3.5, and scalability issues in Section 4.3.4. Section 4.4 provides a short conclusion with respect to the chapter's primary accomplishments.

4.2 The Test Environment

To begin, we note that all evaluations are conducted with the Sidera engine running on a single Linux workstation. Though Sidera is a fully parallelized system, we have used a single node test environment for two reason. First, our new 17-node Linux cluster is not yet available for testing. Second, because the query and caching software stack is designed to work independently on each node of the parallel machine, single node testing is in fact more useful and more intuitive in the current context. In terms of the test platform itself, it is a Linux-based workstation with 1 GB of main memory and a pair of 1.8 GHz Intel processors. Disks are standard 40 GB drives. All software components have been implemented using C++, STL (the Standard Template Library), and the MPI communication functions (even though we run the tests on a single node, MPI libraries are sometime utilized).

Data sets are generated using a custom data generator developed specifically for the Sidera environment. We note that while our data generator provides a mechanism for generating skewed data distributions, we have not used them in this case. Instead, values are randomly generated and uniformly distributed. The reason for using the simpler distribution is that our focus in this environment is the study of the hierarchy mapping mechanisms themselves and data skew adds little to this understanding (though, of course, it is quite important in other contexts). With respect to the data sets, we first generate a multi-dimensional Fact Table (the dimension count varies with the particular test), with cardinalities arbitrarily chosen in the range 2–500. Depending on the test involved, row counts typically vary from 100,000 to 10,000,000 records. The primary fact tables are then used to compute fully materialized data cubes containing hundreds of additional views or *cuboids*. For example, a

10-dimensional input set of 1,000,000 records produced a data cube of 1024 views and approximately 120 million total records. Once the cubes are materialized, we index the tables using the r-tree mechanism provided by the core Sidera engine. Finally, in terms of the hierarchies themselves, we create a mix of hierarchy depths, from two to six levels, with the latter representing the maximum depth that we are likely to encounter in practical settings.

Because individual millisecond-scale queries cannot be accurately timed, we use the standard approach of timing queries in batch mode. In our case, an automated query generator constructs batches of 1000 range queries against several kinds of hierarchies (e.g. Symmetric Strict Hierarchies, Ragged Strict Hierarchies, etc.), in which high/low ranges are randomly generated for each of k dimensions, randomly selected from the d-dimensional space, $k \leq d$. Sort orders are also randomly determined. We note that this form of query generation actually overestimates query response time since users typically query low-dimensional views that can be easily visualized. In the succeeding tests, five batches of queries are generated and the average run-time is computed for each plotted point.

4.3 Experimental Evaluation

Throughout this section we will look at a series of query tests, each intended to highlight a specific aspect or component of our methods. Specifically, we present the experimental evaluation of the hMap, xMap, and nMap algorithms. We begin with an assessment of the computational overhead introduced by the traversal of the mapping data structures.

4.3.1 Evaluation of Hierarchy Overhead

Recall that symmetric strict hierarchies and ragged strict hierarchies are supported by the hMap and xMap respectively. In addition, we handled non-strict hierarchies with the nMap extension. The processing overhead associated with our approaches is the run-time performance penalty incurred by the mapping of queries to/from the base attribute. It is therefore important to evaluate the performance of queries associated exclusively with the base attribute versus those that are free to access arbitrary hierarchy levels.

hMap Overhead

In this case, dimension hierarchies are defined on the base data cube so as to represent the symmetric strict form. (Note that this is done by way of hierarchy tables that are read and processed by the Hierarchy Manager at start up). We then use our query generator to generate batches of 1000 symmetric strict hierarchical queries and 1000 non-hierarchical queries on the same dimensions. The hierarchical queries have ranges randomly defined on one or more levels of the dimension hierarchy. The hMap method is then used to (automatically) answer queries of this type. In terms of the "non-hierarchical" queries, we are referring to queries whose ranges have been restricted to the base attribute. Note that the queries can access any of the 1024 views in the cube; the query engine dynamically selects the most cost effective target.

Figure 4.1 provides the test results. Here, we present the total response time for hierarchical versus non-hierarchical queries. The graph shows the degree of overhead that hierarchical transformation introduces by virtue of its use of the hMap structure. Results are shown for initial fact tables of 100K, one million, and ten million

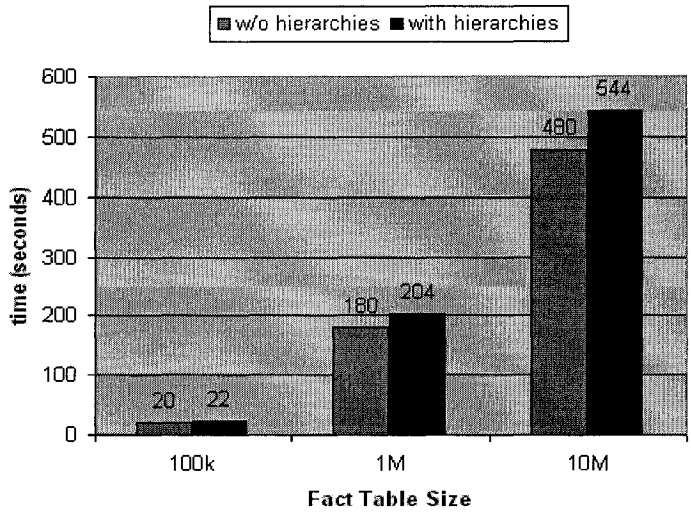


Figure 4.1: Comparison of symmetric strict hierarchical queries versus non-hierarchical queries for three different cube sizes.

records. Note that in this test, the fact table (and associated data cube) contains 10 dimensions, each of which possesses its own hierarchy. In all three cases, total overhead averages less than 12% relative to the non-hierarchical case. It is important to place these results into context. Though we are evaluating our current results in a single node environment, the parallel Sidera query engine is capable, for example, of resolving approximately 100 queries per second for fact tables of one million records (the second bar in the histogram). Even if we were to increase the size of the input set, the difference in cost for the end user is likely to be imperceptible.

xMap Overhead

When ragged hierarchies are present, the xMap data structure is utilized. In this case, we re-define the hierarchical model for the cube environment so as to create ragged dimension hierarchies. Again, we use our query generator to generate batches of 1000

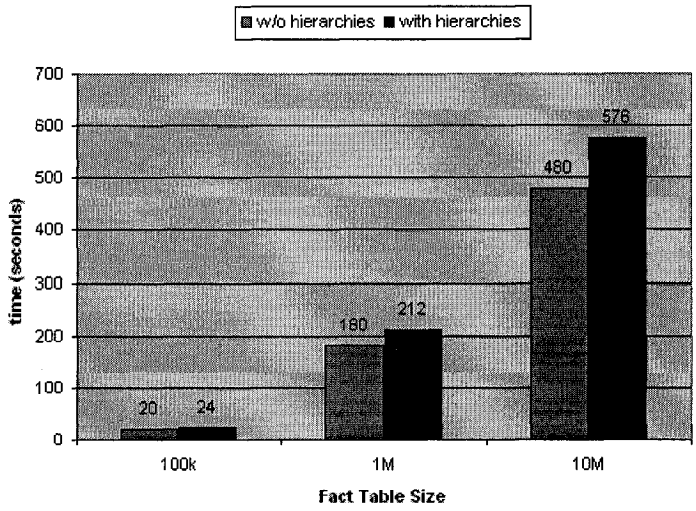


Figure 4.2: Comparison of ragged strict hierarchical versus non-hierarchical queries for three cube sizes.

ragged strict hierarchal queries, with attribute ranges selected at random levels in the hierarchy. The non-hierarchial queries are defined as per the method described in the previous section.

Figure 4.2 provides the total response time for ragged strict hierarchical versus non-hierarchical queries. Results are displayed for 100K, one million, and 10 million records. Recall that the xMap data structure is more sophisticated than the hMap. Of particular significance is the fact that data structure access requires pointer traversal, while the hMap relied mainly upon simple binary searches through static arrays. As a result, we would expect the overhead to increase slightly. That being said, the graph illustrates a relatively modest performance penalty as the overhead grows from the 12% average of the hMap to approximate 17%. Again, we note that in a practical setting an increase in compute time of less than one fifth on average for a single query is likely to be insignificant to users of the system.

nMap Overhead

Finally, we turn our attention to the nMap, the data structure used in resolving queries on non-strict hierarchies. In this case, the dimension hierarchies were redefined so as to include a combination of non-strict symmetric and non-strict ragged hierarchies. Moreover, as we have seen in Chapter 3, we still use the idea of the xMap structure between levels to define one-to-many relationships, while the nMap structure is used to implement the bridge table model for the many-to-many relationships. Again, we use our query generator to generate batches of 1000 queries, in both hierarchical and non-hierarchical form.

Figure 4.3 illustrates the result, in terms of performance overhead, for the two alternate query forms. The results demonstrate an overhead of about 39% on average, a factor of two increase over the pure xMap structure. This is in keeping with the architectural model of the nMap, which must cope with a significant increase in the degree of information required in order to represent many-to-many relationships, as well as the non contiguous nature of parent locations. As to whether the performance overhead would be noticeable to the end users, it is difficult to provide a definitive answer. Given the ability of the Sidera server to provide sub-second response time, even for larger data sets, a two fifths increase in response time relative to the non-hierarchical form would probably not be an issue. For a less efficient query engine, that might not be the case. Nevertheless, we can say that the overhead is certainly modest relative to the power and flexibility that the new methods provide.

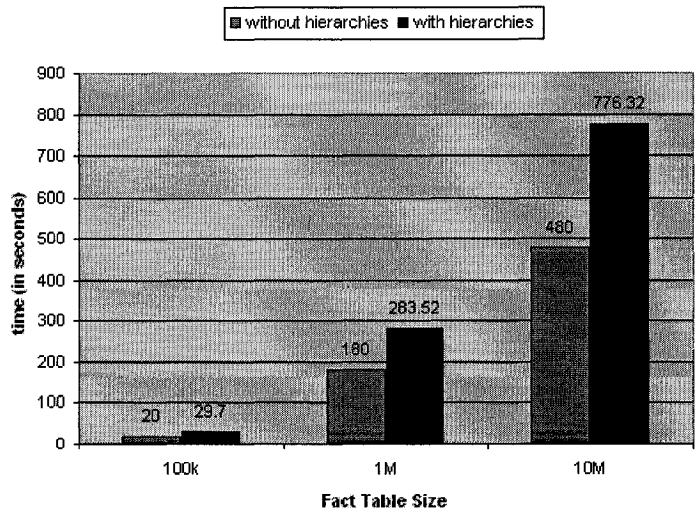


Figure 4.3: Comparison of non-strict ragged and non-strict symmetric hierarchical queries versus non-hierarchical queries for three cube sizes.

4.3.2 hMap and xMap Versus Sort Merge Join

In the absence of the kind of data structures that we have described in this thesis, options for hierarchical dimension processing are quite limited. For strict hierarchies, the standard technique is to redundantly store hierarchy information in the dimension tables of the star schema. A product record, for example, would contain information about its brand and category associations. Alternatively, a Snowflake Schema would normalize this information into three distinct tables. At query time, the SQL query engine would perform a *join* operation on the Fact table and the required dimension tables so as to properly resolve the hierarchical relationships for each record selected from the Fact table.

To provide this comparison, we have extended Sidera so as to support the standard Sort Merge operation. Briefly, the Sort Merge selects appropriate records from both

tables, sorts the intermediate records on a shared column, then merges the two to form the final result. Clearly, this is an expensive operation relative to the mapping tables that do not require access to the dimension tables at all.

In the following two sections, we look at the impact of both Fact table size and dimension count in terms of the Sort Merge technique.

Fact Table Size

In this section, we compare our methods against the traditional Sort Merge join used in many relational database systems. In this case, we create 10 strict dimension hierarchies made up of an even mixture of symmetric and ragged forms. As always, we employ batches of 1000 queries, this time in hierarchical form only. The direct comparison will, of course, evaluate the hMap and xMap against the Sort Merge technique on the same query batches.

Figure 4.4 shows the running time for data sets ranging in size from 100,000 records to 10,000,000 records. As expected, the introduction of a sort and merge phase for each query introduces a substantial penalty during query resolution, in this case approximately an order of magnitude. There are two cautionary points that must be made with respect to a direct interpretation of the results. First, our relatively simple Sort Merge is unlikely to be as efficient as an optimized commercial RDBMS sort. Second, the use of efficient database caches would also improve the run-time of the Sort Merge method. Nevertheless, the Figure graphically illustrates the impact of a reliance upon multi table join operations in order to process dimension hierarchies.

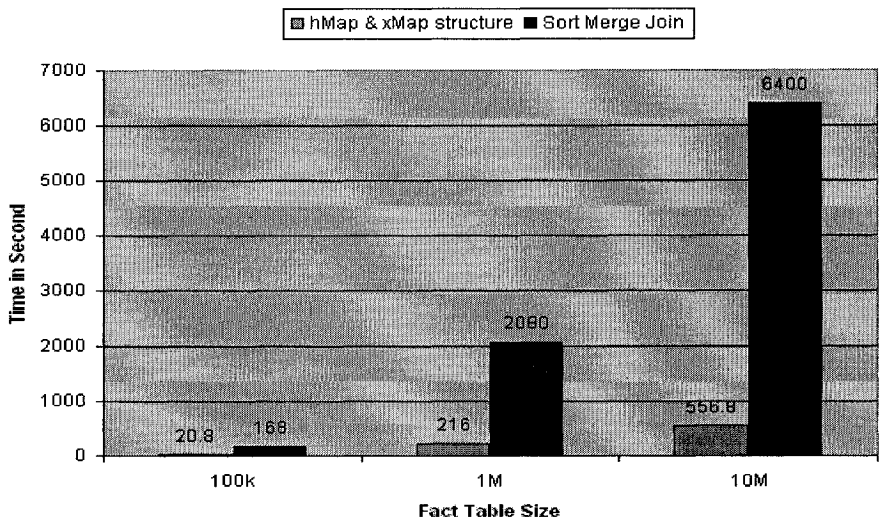


Figure 4.4: Comparison of resolving strict hierarchical queries using both the hMap and xMap versus the Sort Merge Join for the three cube sizes.

Dimension Count

In addition to the impact of data set size, we also look at the impact of an increase in dimension count on query resolution. Recall that with the Sort Merge technique, an independent sort must be performed for each hierarchical dimension since the join columns are distinct. As a result, we would expect to see a deterioration in performance as the dimension count increases. For the current test, we again generate batches of 1000 strict hierarchical queries, this time holding the data set size constant at one million records. Figure 4.5 illustrates the performance of both our approach and the Sort Merge technique as the dimension count increases to a maximum of ten, a number indicative of the maximum number one might expect to see in practical environments. Perhaps not surprisingly, we observe that the query resolution time for batches slowly deteriorates across the range, ultimately producing a factor of nine

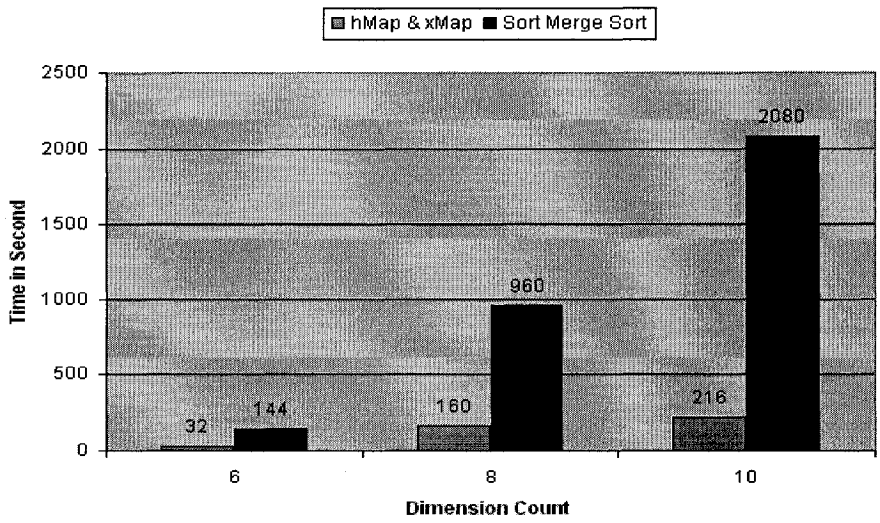


Figure 4.5: Comparison of resolving strict hierarchical queries using both the hMap and xMap versus Sort merge, as a function of dimension count.

increase at 10 dimensions. Again, the same cautions with respect to the previous test apply. Still, the benefit of compact, in-memory translation structures for such a crucial OLAP operation is undeniable.

4.3.3 The hMAP and xMap versus commercial implementations

As noted above, we expect commercial implementations of the Sort-Merge join to outperform our own. Consequently, we provide one more test in this section to evaluate our hierarchical framework against a join-based commercial DBMS. In this case, we use Microsoft SQL Server 2005. Here, we build a 10-dimensional fact table with 100,000 records, with the cardinality for each dimension varying between 2 and 500. Dimensions are arbitrarily chosen as strict symmetric hierarchies and strict ragged hierarchies (hMap and xMap compatible). Note that in this test we do not

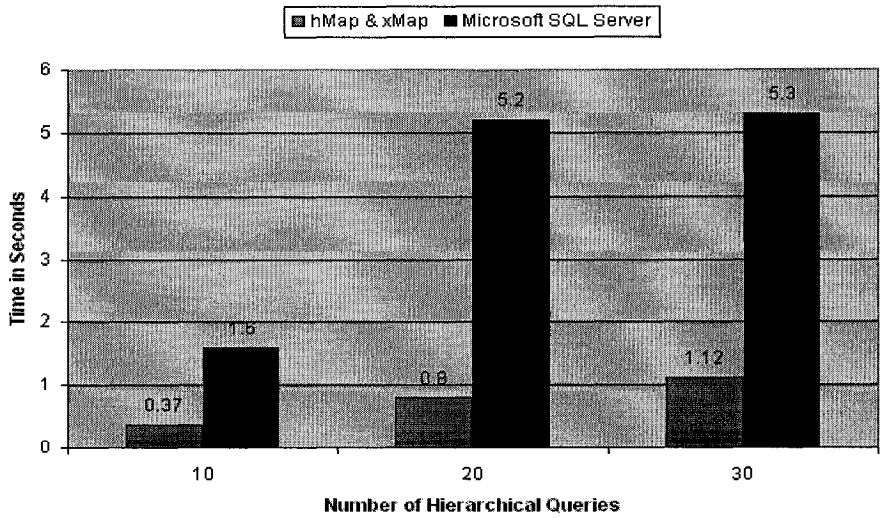


Figure 4.6: Comparison of resolving hierarchical queries using our approach in modelling dimensions hierarchies against Microsoft SQL server.

materialize any aggregate group-bys. In other words, all tests are conducted simply against the fact table and the dimension tables in order to allow a fair comparison of the results. (Our own system does not require access to the dimension tables, of course.) In terms of the queries, we first generate small batches using our own query generator. Then, each of these Sidera queries is translated by hand into the appropriate SQL syntax so that they can be executed on the SQL Server DBMS. Because of the labour-intensive nature of this task, we cannot use batches of 1000 queries, instead relying on batches of 10 to 30 queries.

Figure 4.6 provides the test results. Here, we present the total response time for strict hierarchical queries executed by our query engine versus strict hierarchical queries answered by Microsoft SQL server. As expected, the commercial DBMS shows improved response time relative to our own Sort Merge implementation. However, across all three batch counts, a dramatic overall performance difference relative to our

mapping model is still apparent. Specifically, the hMap/xMap model is approximately 5 times faster than the one based upon the multi-join mechanism typical of current data warehouse implementations.

4.3.4 Scalability

In production environments, it is quite likely that OLAP users will be accessing systems that are larger than the ones that can be conveniently tested in academic settings. As a result, it is important to provide some understanding of performance as core parameters grow. Our scalability assessment begins with a look at performance patterns as the number of records increases from 100K to ten million. We use a 10-dimensional fact table, and we define the dimensions so as to contain symmetric strict hierarchies, ragged strict hierarchies, symmetric non-strict hierarchies, and ragged non-strict hierarchies. In other words, the query engine utilizes all three mapping forms, selecting the most appropriate for each particular hierarchy.

Figure 4.7 shows the execution time for hierarchical queries as a function of data cube size. As can be seen in the figure, an increase in the number of records in the fact table is associated with a linear increase in execution time. In fact, this is as expected since the worst case bound of a given mapping model is effectively fixed. For example, with the xMap, translation time is bounded by the number of levels in the hierarchy. As a result, the maps introduce an essentially constant overhead for the translation of individual columns. The performance curve as a function of data set size is therefore unaffected.

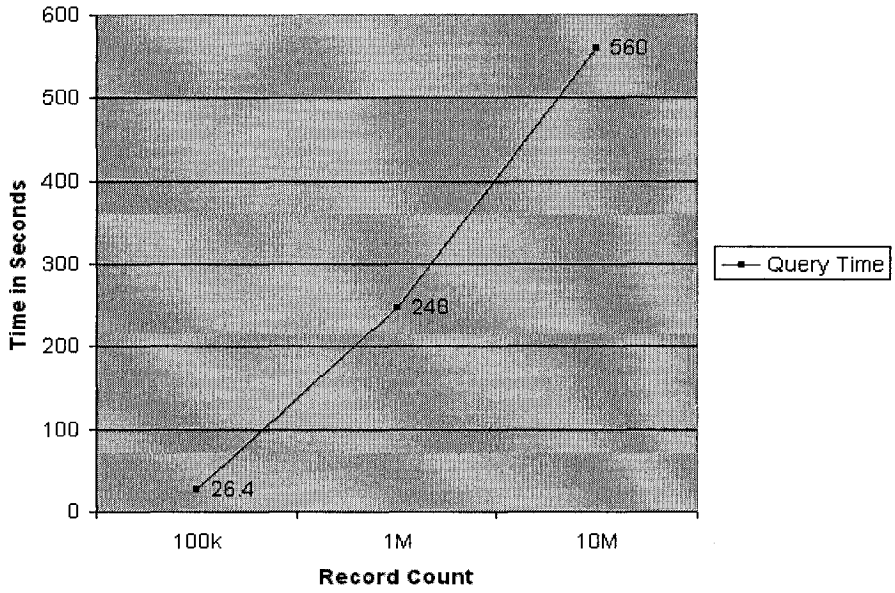


Figure 4.7: Execution time for hierarchical queries as a function of the data cube size.

Hierarchy Depth

Because the cost of individual hierarchy translation operations is ultimately bounded by the depth of the tree, we also chose to look at how the performance of the xMap changes with an increase in hierarchy depth (recall that the hMap has a fixed height of three and the nMap is typically used in conjunction with the xMap). For this test, we create a 10 dimensional fact table with one million records. We then set the depth of all 10 dimension hierarchies to be of size two, then four, then six. Note that by "depth", we mean the number of levels in the hierarchy. In practice, a depth of six should approach the practical limit for a single hierarchy. Again, we generate a batch of 1000 hierarchical queries that are free to access ranges at any depth of the hierarchy map.

Figure 4.8 shows the results for the three dimension depths. We may note two

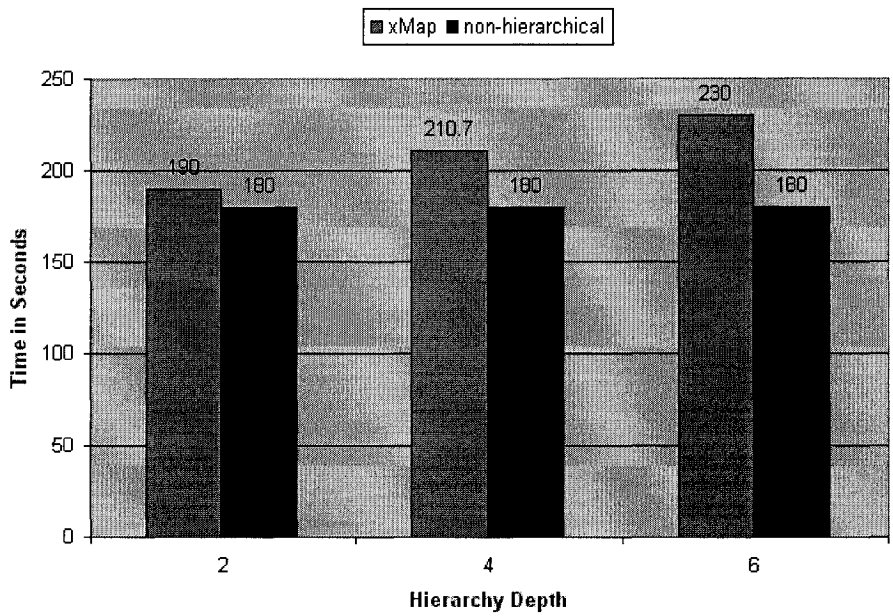


Figure 4.8: Processing overhead as a function of hierarchy depth

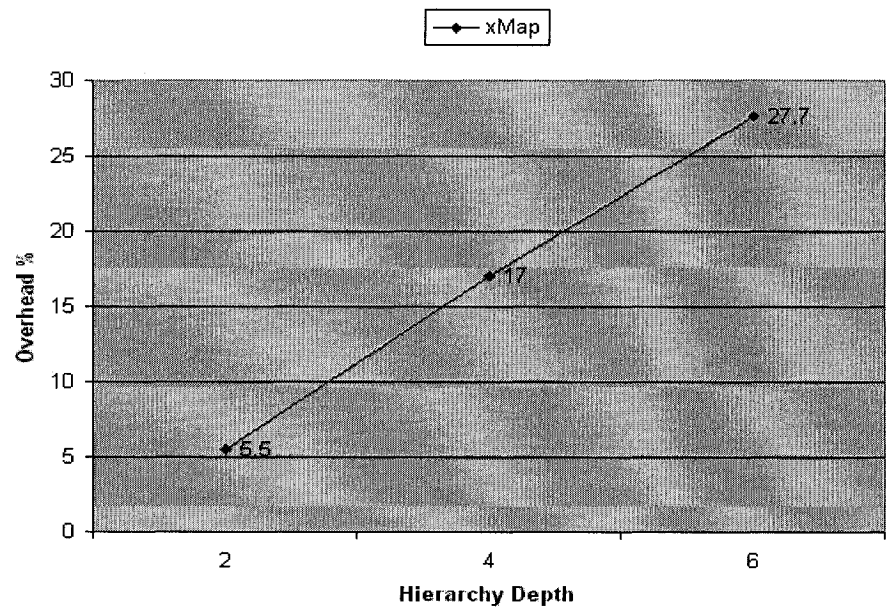


Figure 4.9: Rate of overhead growth as a function of hierarchy depth.

things. First, the impact of hierarchical processing is fairly small relative to the total query time. This is the case since other costs such as aggregation and I/O are far more significant (and constant for a given query batch). Second, as depicted in Figure 4.9, the growth in overhead as a function of hierarchy depth is essentially linear. Specifically, overhead from two to six levels grows from approximately 5% to 27%. Note that when the hierarchy depth is two, then $l_{max(A_i)}$ is one. When we increase the depth to 4, then $l_{max(A_i)}$ is 3. In turn, xMap overhead increases by a factor of three. When we increase the depth from two to six, we have $l_{max(A_i)} = 5$, and an overhead penalty of approximately the same amount. This is a very encouraging result, particularly when we consider that having 10 six-level hierarchies is probably a significant overestimate of the typical OLAP scenario.

4.3.5 Non-strict Hierarchies: nMap Structure versus the alternative

In this section, we compare the performance of our nMap mechanism in resolving non-strict hierarchical queries relative to the approach proposed by Pedersen[27]. Here, the objective is to transform the non-strict hierarchies into many strict hierarchies. Suppose that we have a 4-dimensional fact table, where one dimension (i.e. employee) contains a non-strict hierarchy. We therefore have to convert this non-strict employee hierarchy into several strict hierarchies where the number of these strict hierarchies is proportional to the number of many-to-many relationships between consecutive levels. Figure 4.10 shows the symmetric non-strict hierarchy where employees may work in several sections. For simplicity, we assume that the non-strict hierarchy has only one many-to-many relationship between the leaf level and its parent level.

Figure 4.11 shows the fact table connected to the relevant dimension table by the

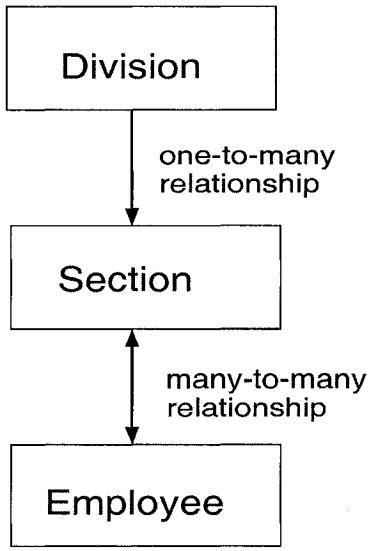


Figure 4.10: A symmetric non-strict hierarchy (Employee).

appropriate key values (the other dimensions are omitted for clarity). The focus of analysis in this figure is employee salary. Using the Pedersen technique, we transform the non-strict employee dimension into two independent strict dimensions as shown in Figure 4.12. Note that the focus of analysis has been changed now to "employee salary by section" instead of just employee salary. Moreover, the fact relation in Figure 4.12 contains more detailed data than that of Figure 4.11. In the former, there is just one row for each section in which an employee works; in the latter there is one row per employee. Thus, the fact table will include more rows using the Pedersen technique.

Given the preceding description, we now compare our nMap model to the Pedersen proposal. We use a 5-dimensional fact table, with non-strict hierarchies chosen for each dimension. The record count of the fact table is 1K, 10K and 100K (the smaller size is used due to the growth of the Pedersen-based fact table). We transform the 5 non-strict hierarchies into simple hierarchies in order to accurately represent the

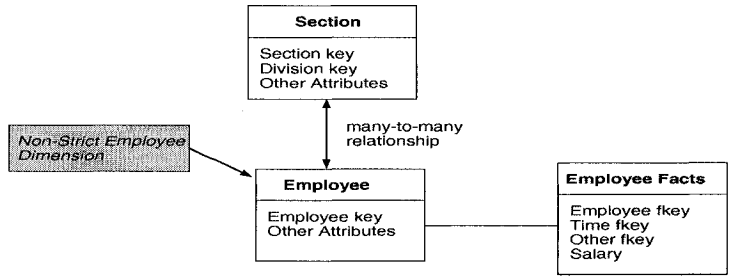


Figure 4.11: A fact table connected with non-strict hierarchy (Employee).

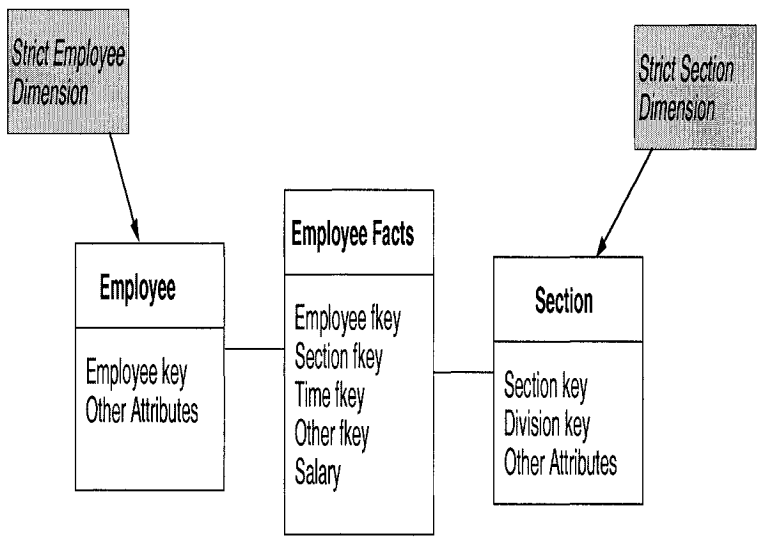


Figure 4.12: Transformation of a non-strict hierarchy (Employee) into a strict hierarchy.

FT Size (records)	# of Non-strict Hx	FT Size	# of Strict Hx
1K	5	100K	10
10K	5	1M	10
100K	5	10M	10

Table 4.1: The 5-dimensional non-strict fact table and its correspondence in strict hierarchies.

Pedersen method. After transformation, we must ensure that we have the same data in both fact tables. Therefore, we transform the 5 non-strict hierarchies and the initial fact table into 10 strict hierarchies and a much larger final fact table. Table 4.1 summarizes the size of the 5-dimensional fact tables with non-strict dimensions and the corresponding 10-dimensional fact tables after the transformation has taken place.

Figure 4.13 provides the test results. Here, we present the total response time for non-strict hierarchical queries modelled by the nMap versus the strict hierarchical transformation of the Pedersen method. Observe that the nMap structure is 15 to 20 times faster than the technique proposed by Pederson. Simply put, while we can verify that the Peterson method does in fact produce the appropriate result, it is complex to design and scales very poorly relative to the mapping structures we have presented.

4.3.6 Multi-dimensional Caching

In practice, OLAP queries tend to be iterative in nature. Users often define an initial exploratory query and then gradually refine the scope of the original query to obtain the desired result. Drill down, roll up, slice and dice, and pivot form the basis of such cube traversals. In the absence of a multidimensional, hierarchy-aware caching framework, the cost of Five Form processing is likely to grow significantly. On average, Sidera's caching design reduces query resolution time by 10 – 20% (for 1000

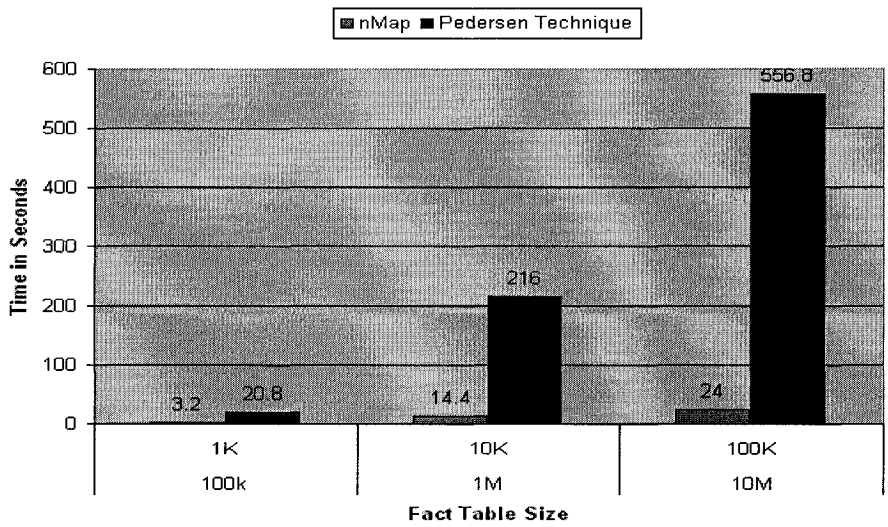


Figure 4.13: Comparison of resolving non-strict hierarchical queries using nMap against Pederson technique three different cubes as showed in Table 4.1.

query batches) on the current system, depending upon the workload of the operating system and the sizes of its own disk caches (in the current case, this was a lightly loaded system using the standard Linux defaults). Because this is an OLAP or cube-based cache, it is not at this point terribly meaningful to compare these results to other database systems that typically employ non-cubic caches that are optimized for other purposes. Moreover, we expect that our cache manager will become significantly more valuable when it processes partial hits.

It is informative, however, to examine the *cache hit ratio* for the current Sidera system. Specifically, we would like to know how often cube-oriented queries reprocess the same portion of the multi-dimensional space, given a query batch of modest size. We note, of course, that iterative, hierarchy-based queries (e.g., roll up and drill down), are by definition drawn exclusively from cache-managed memory. Even for *isolated* range queries, however, the current cache framework is extremely effective.

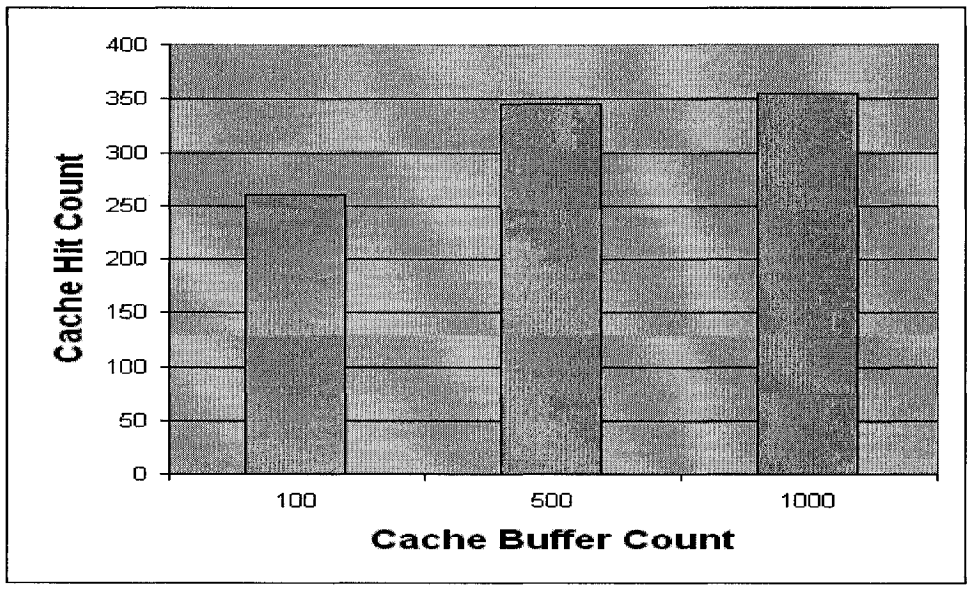


Figure 4.14: Comparison of cache hit rates for three buffer counts and batches of 1000 queries.

In Figure 4.14, we see the effect of caching on the cube generated from the 1M-record fact table. Again, batches of 1000 queries and the average of five runs are used. For batches of 1000 queries, the graph shows the average hit rate as the number of available buffers increases from 100 to 1000. Specifically, the rate increases from 265 per 1000 queries to 355. Interestingly, a doubling of the buffer count from 500 to 1000 does relatively little as the 500-buffer model is able to achieve an average hit rate of 340. Again, we note that in practice the actual hit rate will be far higher than this since virtually 100% of the canonical OLAP query forms will be resolved directly from previously cached results.

4.4 Conclusions

In this chapter, we have provided experimental results that assess the ability of the query engine to efficiently support queries in the presence of several kinds of hierarchies. Experimentally, our results support the design decisions that we have made. Test results demonstrate that real time processing overhead is likely to be imperceptible to the end user. Moreover, we show that our mapping methods compare very favorably to the techniques that are utilized in current systems, as well as proposals from the academic literature. Scalability, in terms of record size and hierarchy depth, is also assessed. Finally, we provide evidence that our integrated, hierarchy-aware caching subsystem has the potential to significantly boost run-time query performance.

Chapter 5

Conclusions and Future Work

5.1 Summary

Our objective in this thesis has been the creation of data structures and access methods for the efficient "real time" manipulation of realistic attribute hierarchies. The use of this hierarchy-aware framework represents a significant improvement over the awkward and inefficient techniques often employed in contemporary data warehouses. Furthermore, our structures for hierarchical attributes have been integrated into the Sidera data warehousing server architecture, demonstrating the viability of this approach on a reasonably sophisticated and comprehensive prototype. Processing functionality is identical to that provided for simple, non-hierarchical attributes. We expect that in most cases, minimal processing overhead is generated and the difference in cost between hierarchical and non-hierarchical processing would likely not be perceptible to end users. This is in stark contrast to the current methods which introduce far more substantial query overhead.

To summarize, we have focused on the following targets:

1. **The hMap**. We have created a powerful structure we call the hMap for modelling strict symmetric hierarchies. To do so, we exploit the notion of *attribute*

linearity. This approach permits translations between arbitrary levels of the dimension hierarchy in a manner that requires very low memory resources. Justification for the hMap technique is provided by way of analysis and experimentation.

2. **The xMap**. We have presented an extended structure known as the xMap for modelling both symmetric strict hierarchies and ragged strict hierarchies. The xMap still supports the notion of attribute linearity. The new data structure does not require significant additional memory, instead relying on efficient mapping and transformation services that can be cost-effectively applied at run-time. Extensive testing of this structure demonstrates fast resolution of hierarchical queries.

3. **The nMap**. Real world data warehouse environments must support more than than just strict hierarchies. However, the problem of modelling non-strict hierarchies is not well studied in the literature. We provide a structure known as the nMap that can handle the non-strict hierarchies by building on the idea of a bridging table. Despite the complexity of many-to-many relationships, experimental results demonstrate only modest performance overhead on non-strict hierarchical queries.

4. **Caching Hierarchical ROLAP Queries**. We have provided a hierarchy-aware caching model to further optimize query response time. In short, the query engine transparently manipulates cached hierarchical content to further refine previous hierarchical user queries. Consequently, the hierarchy-aware, multi-dimensional caching framework provides direct support for each of the

fundamental OLAP query operations.

5.2 Future Work

The research described in this thesis represents the foundation for the development of a robust framework for the management of ROLAP hierarchies. Below we identify a number of possible projects or research themes that would significantly extend the functionality of the current design:

1. **Extended Hierarchical Models**. Our approach can be used to model four distinct kinds of hierarchies. However, in the real world we find several additional types of hierarchies (e.g., Parallel hierarchies, Multiple Hierarchies, etc.). It would be important to extend our approach to deal with these kinds of dimension hierarchies.

2. **Support of MDX-Style Queries**. We have built our mapping structures so as to support MDX-style query statements. That being said, we currently utilize a very simple form of this syntax. To exploit the full potential of our map structures, the system should be extended to include a more realistic *map-aware* query language.

3. **Hierarchy Definition Language**. In the SQL world, we often talk in terms of the Data Definition Language (DDL) and the Data Manipulation Language (DML). Though both are SQL, one is used to create RDBMS structure, while the other is used to query them. Likewise, in addition to the query functionality discussed in the previous item, the new query language should make it relatively easy to define or model complex hierarchical relationships so that the mapping

structures can be effectively utilized.

4. **Partially cached queries**. As noted in Chapter 3, our hierarchy aware caching model does not currently answer hierarchical queries that only partially intersect the current cache *bounding box*. Clearly, there would be great advantage to using partial cache matches in order to reduce the number of blocks that must be retrieved from disk.

5.3 Final Thoughts

The research presented in this thesis provides a powerful framework for hierarchical query resolution in OLAP environments. In contrast to other suggestions in the literature, it does not rely upon expensive multi-table joins that ultimately provide limited functionality. We have discussed the motivation, implementation, and evaluation of our results and have emphasized their practicality in real worlds settings. Given the significance of the hierarchical modelling problem in ROLAP environments, we are convinced that our research represents a meaningful addition to the literature in this area.

Bibliography

[1] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[2] L. Cabibbo and R. Torlone. Querying multidimensional databases. *Proceedings of the 6th DBLP Workshop*, pages 253–269, 1997.

[3] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[4] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel ROLAP datacube construction on shared nothing multi-processors. *International Parallel and Distributed Processing Symposium*, 2003.

[5] E. Codd, S. Codd, and C. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E.F. Codd and Associates, 1992.

[6] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

[7] F. Dehne, T. Eavis, and A. Rau-Chaplin. The cgmCUBE project: Optimizing parallel data cube generation for ROLAP. *Journal of Parallel and Distributed Databases*, 2005. To appear.

[8] B. Dinter, C. Sapia, G. Hofling, and M. Blaschka. The OLAP market: State of the art and research issues. *ACM First International Workshop on Data Warehousing and OLAP*, pages 22–27, 1998.

[9] E.Zimanyi E. Malinowski. Olap hierarchies: A conceptual perspective. *Advanced Information Systems Engineering, 16th International Conference, CAiSE*, 2004.

[10] E.Zimanyi E. Malinowski. Hierarchies in a conceptual model: From conceptual modeling to logical representation. *Data KNowledge Engineering*, 2005.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, 1984.

[13] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2000.

[14] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.

[15] C. Hurtado, A. Mendelzon, and A Vaisman. Maintaining data cubes under dimension updates. *Proceedings of the IEEE Interantional Conference on Data Engineering*, 1999.

[16] W. Inmon. *Building the Data Warehouse.* John Wiley, 1992.

[17] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava. What can hierarchies do for data warehouses? *The VLDB Journal*, pages 530–541, 1999.

[18] Ralph Kimball and J. Caserta. *The Data Warehouse ETL Toolkit.* John Wiley and Sons, 2004.

[19] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit.* John Wiley and Sons, 2002.

[20] H. Lenz and A. Shoshani. Summarizability in OLAP and statistical data bases. *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, pages 132–143, 1997.

[21] Volker Markl and Rudolf Bayer. Processing relational olap queries with ub-trees and multidimensional hierarchical clustering. *DMDW*, 2000.

[22] Analysis Services: MDX, 2006. http://msdn.microsoft.com.

[23] http://msdn2.microsoft.com/en-us/library/ms123402.aspx.

[24] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[25] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[26] T. Niemi, J. Nummenmaa, and P. Thanish. Logical multidimensional database design for ragged and unbalanced aggregation hierarchies. *International Workshop on Design and Management of Data Warehouses, DMDW 2001*, pages 1–8, 2001.

[27] T. Pedersen, C. Jensen, and C. Dyreson. A foundation for capturing and querying complex multidimensonal data. *Information Systems Journal*, 26(5):383–423, 2001.

[28] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.

[29] Hans Sagan. Space-filling curves. *Springer-Verlag*, 1994.

[30] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[31] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1990.

[32] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.

[33] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Hierarchical dwarfs for the rollup cube. *DOLAP 03: Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, pages 17–24, 2003.

[34] Abhishek Sugandhi. Data warehouse design considerations. *M. Tech. Course Seminar Report.*

[35] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.

[36] Chendong Zou, Betty Salzberg, and Rivka Ladin. Back to the future: Dynamic hierarchical clustering. *ICDE*, 1998.